Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Mojca Komavec

Avtomatska izgradnja modela odvisnosti med komponentami IT

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Marko Bajec
Somentor: Assoc.Prof. Dipl.-Ing. Dr.techn. Denis Helic

Ljubljana, 2017

University of Ljubljana
Faculty of Computer and Information Science

Mojca Komavec

Automated Generation of IT Component Dependency Models

master's thesis

the 2nd cycle master's study programme
computer and information science

Supervisor: prof. dr. Marko Bajec
Co-supervisor: Assoc.Prof. Dipl.-Ing. Dr.techn. Denis Helic

Ljubljana, 2017

Povzetek

Naslov: Avtomatska izgradnja modela odvisnosti med komponentami IT

Tehnologija in infrastruktura IT postajata zelo pomembna pri velikih podjetjih, saj ponujata hiter odziv in razrešitev ponavljajočih se problemov, napak in odkritje osnovnih vzrokov le-teh. Za avtomatizacijo teh procesov se podjetja zanašajo na pristop, imenovan identifikacija izvora napak. Ena izmed komponent je modul za odkrivanje komponent, ki med drugim vsebuje tudi informacijo o odvisnostih med komponentami IT. V tej nalogi se osredotočamo na avtomatsko gradnjo grafa odvisnosti med komponentami IT iz konfiguracijskih parametrov. Podatke analiziramo tako, da najprej najdemo odvisnosti med gostitelji, nato pa med posameznimi komponentami IT. Vsaki odvisnosti z algoritmom strojnega učenja dodelimo verjetnost, da obstaja. Prav tako pokažemo, da je naš pristop hitrejši in bolj natančen kot naiven pristop, ki primerja vsak konfiguracijski parameter z vsakim. Naloga vsebuje tudi obsežno evalvacijo na resničnih podatkih. Evalvacija upošteva tranzitivno lastnost, ki velja za odvisnosti, in specifične lastnosti, ki veljajo pri analizi identifikacije izvora napak. Z rezultati evalvacije pokažemo, da naš predlagan algoritem doseže 90% priklic in 100% natančnost pri odkrivanju odvisnosti med generičnimi komponentami IT.

Ključne besede

analiza podatkov, preslikava odvisnosti med komponentami, odvisnosti med konfiguracijami, analiza odvisnosti, identifikacija izvora napak, usmerjeni graf

Abstract

Title: Automated Generation of IT Component Dependency Models

Large enterprises are heavily relying on IT technology and infrastructure, which strives to quickly respond and remediate occurring problems, faults, and identify the underlying root causes. To automate this process, the enterprises rely on root cause analysis approach. One of the components is a component discovery module, which also provides information about the dependencies between IT components. In this thesis, we focus on building an IT component dependency graph from granular configuration data automatically. We analyze the configuration data in order to first infer the dependencies between hosts, and secondly, to find the dependencies between IT components. Furthermore, we assign each dependency a likelihood that it exists with a supervised machine learning algorithm. We show that our approach is much faster and accurate compared to the naive approach, which compares configuration parameters to each other. Moreover, we provide an extensive evaluation on the real dataset, where the evaluation takes into account the transitive property of dependencies, and specific properties of the root cause analysis. The evaluation results show that our proposed algorithm reaches 90% recall and 100% precision for discovering dependencies between generic IT components.

Keywords

data analysis, component dependency mapping, configuration dependencies, dependency analysis, root cause analysis, directed graph

Acknowledgments

This thesis could not have been done without the generous help, assistance, and participation I received from many people. I am very grateful to each and every one of them for their contributions. However, there are some people I would like to express my deepest gratitude to.

I would like to first thank my non-official mentor, Boštjan Kaluža. Thank you for pushing me when I most needed it, for the guidance and ideas, that helped outline this thesis. Thank you for the corrections of the first drafts, for unselfish help and encouragement that helped me finish this thesis. Thank you.

I would also like to thank all my Evolven colleagues. Thank you for giving me the opportunity to work in such inspiring team and the possibility to extend the work into my master thesis.

I would like to express my gratitude to my mentor, Marko Bajec, for providing me quick feedback and the ideas about the evaluation part. I would also like to express my gratitude my co-mentor, Denis Helic. Thank you for showing great interest and giving me some great ideas.

In addition, I would like to thank the Erasmus programme, for giving me the scholarship and the opportunities for studying abroad, especially at TU Graz, where I had the ability to join the double-degree programme. It has been a great experience in gaining knowledge both, professionally and personally.

I would also like to thank my nono, Aleš Komavec, for taking the time to proofread the English part of the thesis.

Last but not least, I would like to thank my parents, Viktorija and Tadej, and my boyfriend, Benjamin. Thank you for your love, continuous support, and encouragement throughout my years of study. It would not have been the same without you.

Mojca Komavec, 2017

To my grandmother Viktorija.
For always believing in the importance of education and hard work.

Contents

List of used acronyms

acronym	meaning
SLA	Service Level Agreement
IT	Information Technology
ITIL	Information Technology Infrastructure Library
ITO	Information Technology Operations
ITSM	Information Technology Service Management
RFC	Request For Change
ITOA	Information Technology Operations Analytics
RCA	root cause analysis
APM	Application Performance Monitoring
CMDB	Change Management Database
IP	Internet Protocol
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
CPU	Central processing unit
HTTP	Hypertext Transfer Protocol
CI	Configuration Item
RAM	Random-access memory
BIOS	Basic Input/Output System
OSI	Open Systems Interconnection
APM	Application Performance Management
PDM	Probabilistic Dependency Matching

DCG	Direct Cyclic Graph
HDE	Host Dependency Extractor
IDG	Instance Dependency Generator
CIDE	Configuration Item Dependency Extractor
IDE	Instance Dependency Extractor
idf	inverse document frequency
TP	true positive
FP	false positive
FN	false negative

Razširjeni povzetek

I Uvod

Dandanes se velika podjetja zanašajo na podatkovne centre, ki so sestavljeni iz velikega števila strežnikov, kateri nudijo veliko število poslovnih aplikacij končnim uporabnikom, in pripadajočih komponent [41].

Podjetja si prizadevajo za zanesljivost in odzivnost aplikacij, da zagotovijo delovanje storitev skladno z dokumentom SLA. Le-ta predstavlja enega glavnih vidikov operacij IT (ITO). Po definiciji ITO predstavljajo ljudje in upravljalski procesi, ki so povezani z upravljanjem storitev IT (ITSM), s ciljem, da zagotovijo ustrezen nivo storitev z ustrezno kvaliteto po konkurenčni ceni končnim uporabnikom [25]. Kljub temu pa se problemi lahko pojavijo in jih je včasih težko odkriti in razrešiti zaradi kompleksne strukture in velikosti okolja IT. Ker lahko ročno odkrivanje in identifikacija problemov vzame veliko časa in sredstev, podjetja uporabljajo prakso imenovano analitika operacij IT (ITOA). ITOA je praksa nadzorovanja sistema ter zbiranja, procesiranja, analiziranja in razumevanja podatkov iz različnih virov z namenom sprejemanja odločitev in napovedovanja morebitnih problemov [39].

Eno izmed podpodročij ITOA je identifikacija izvora napak (RCA), katerega cilj je najti izvor, ki povroča problem v okolju IT. Na primer, sprememba v nastavitvah požarnega zidu lahko povzroči nedosegljivost storitve uporabnikom zunaj lokalnega omrežja. Ročno voden RCA zahteva strokovno znanje domene in leta izkušenj, zato se ITOA osredotoča na avtomatiziranje RCA [34, 32, 10, 22, 28, 24]. Industrijsko poročilo nakazuje, da se v več kot 85%

primerov incidentov izvor skriva v spremembah okolja IT [26], kar motivira razvoj RCA, ki je osredotočen na spremembe.

V splošnem je sistem RCA zgrajen iz *senzorskega modula*, ki odkrije kršitev SLA ali drug problem z okoljem IT; *modula za odkrivanje komponent*, ki identificira seznam komponent v sistemu IT in odvisnosti med njimi; *modula zgodovine sprememb*, ki vrne seznam vseh sprememb na določenih artefaktih/komponentah; ter *modula sklepanja*, ki poveže incident z najbolj verjetnimi izvori napake.

I.I Formulacija problema

V zadnjem desetletju se je v industriji razvilo veliko zanesljivih orodij za senzorski modul in modul zgodovine sprememb, razvoj modula odkrivanja komponent pa povzroča kar nekaj težav zaradi velikega števila konfiguracijskih podatkov in komponent ter dinamične narave le-teh. Ročno iskanje in vzdrževanje odvisnosti med komponentami ni najbolj smiselno, saj je zanj potrebno strokovno znanje in veliko časa.

Zaradi tega se bomo v tej nalogi osredotočili na izdelavo avtomatske rešitve, ki iz konfiguracijskih podatkov najde odvisnosti med komponentami in se odziva na dinamične spremembe. Tako pridobimo graf odvisnosti med komponentami [33], ki je eden ključnih elementov v RCA, saj omogoča omejitev iskalnega prostora med možnimi kandidati za RCA.

I.II Znanstveni prispevki

Znanstveni prispevki naloge so naslednji. Prvi je formalizacija ogrodja za določanje odvisnosti, s katerim predstavimo ključne komponente, ki jih potrebujemo za gradnjo grafa odvisnosti med komponentami. Drugi znanstveni prispevek je algoritem, ki iz podrobnih konfiguracijskih podatkov in strukture gostitelja določi verjetnost odvisnosti med komponentami. Zadnji znanstveni prispevek predstavljata tehniki evalvacije, ki sta prilagojeni grafu komponentnih odvisnosti, saj upoštevata tranzitivno lastnost odvisnosti. Ena izmed

njih je tudi prilagojena izvoru napak. V kolikor nam je znano, obe tehniki evalvacije nista bili obravnavani pri evalvaciji grafa komponentnih odvisnosti v nobenem sorodnem delu.

V nadaljevanju si bomo najprej pogledali kratek pregled sorodnih del, nato bomo opisali ogrodje za določanje odvisnosti, algoritem za iskanje odvisnosti, opisali bomo rezultate ter zaključili s sklepom.

II Kratek pregled sorodnih del

Med sorodnimi deli, ki opisujejo gradnjo grafa komponentnih odvisnosti, najdemo dve glavni metodi za gradnjo: posredne in neposredne metode [23]. Neposredne metode potrebujejo človeka ali statični program, ki analizira sistemske konfiguracije, podatke o namestitvi in aplikacijsko kodo z namenom, da najde odvisnosti. Neposredne metode lahko uporabljamo le nad specifičnimi sistemi.

Posredne metode delujejo v času izvajanja. Delijo se na vsiljive, delno-vsiljive in ne-vsiljive metode, glede na stopnjo odvisnosti od instrumentacije kode. Vsiljive metode se v celoti zanašajo na instrumentacijo kode, delno- in ne-vsiljive metode pa ne.

Glavne razlike med posrednimi in neposrednimi metodami so sledeče. Neposredne metode potrebujejo znanje o sistemu, medtem ko ga posredne metode ne potrebujejo in so zatorej bolj splošne in lahko delujejo na različnih sistemih. Ni pa nujno, da obe metodi odkrijeta vse možne odvisnosti. Posredne metode ne morejo odkriti odvisnosti med komponentami, ki niso specifične sistemu, neposredne metode pa ne morejo najti odvisnosti med komponentami, ki ne delujejo v času izvajanja.

Med sorodnimi deli obstaja veliko načinov za gradnjo grafa komponentnih odvisnosti, ki lahko sledijo samo eni metodi, ali pa so mešanica različnih.

II.I Posredne metode

Primer vsiljive posredne metode so predstavili avtorji v [14]. Avtorji so razvili sistem, ki najde odvisnosti tako, da označi zahtevke odjemalca, medtem ko le-ti potujejo po sistemu in istočasno išče odvisnosti. Vendar se avtorji v članku niso osredotočili na gradnjo grafa komponentnih odvisnosti; zato tudi niso ovrednotili svoje metode.

Primere delno-vsiljivih posrednih metod so predstavili avtorji v [13, 15, 11]. V [13] so avtorji razvili sistem, ki najde odvisnosti med aplikacijami preko vstavljanja napak in perturbacij. Ovrednotenje svojega sistema so naredili na manjšem, polno funkcionalnem okolju. Dosegli so 99% klasifikacijsko točnost in 97% natančnost. Avtorji so v [15] razvili orodje, ki najde odvisnosti med aplikacijami z analizo prometa, ki poteka med aplikacijami. Orodje so ovrednotili s petimi aplikacijami ter dosegli 99% klasifikacijsko točnost in 21% natančnost. V [11] so raziskovalci razvili pristop, ki najde odvisnosti med komponentami aplikacij z analizo sledi med aplikacijskimi zahtevki in odzivi. Njihov pristop je evalviran z majhno, spletno aplikacijo, kjer so dosegli delež napačno pozitivnih odvisnosti med 21%-29%.

Gradnja grafa komponentnih odvisnosti z uporabo podatkov iz omrežnega prometa je uspešna tudi v industriji, s produktoma ServiceNow [37] in IlumniIt [40], ki pa sta mešanica delno-vsiljive posredne in neposredne metode, ker se zanašajo tudi na konfiguracijske podatke. Oba pristopa najdeta odvisnosti tako med gostitelji kot aplikacijami.

Ne-vsiljive posredne metode se zanašajo na pridobivanje podatkov o zmogljivosti iz sistema [23], z uporabo različnih orodij [19, 32] ali pa rudarijo dnevniške datoteke [38], z namenom iskanja odvisnosti med komponentami. Predpostavka je, da sta dve komponenti odvisni, če se njihova aktivnost zgodi večkrat približno istočasno. Raziskovalci so v [23] zgradili sistem, ki najde odvisnosti med komponentami aplikacij iz podatkov izvajanja. Sistem so ovrednotili z okoljem, zgrajenim iz treh strežnikov. Odvisnosti so bile odkrite s 100% klasifikacijsko točnostjo in natančnostjo med 63% in 100%. Ensel [19] je uporabil nevronske mreže, ki iz časovnih podatkov o obreme-

nitvah zgradijo graf odvisnosti med komponentami. Le-ta je zmožen ugotoviti odvisnosti med gostitelji in aplikacijami. Opisana je le arhitektura brez evalvacije. Avtorji so v [32] našli odvisnosti med komponentami s pomočjo podatkov o obremenitvah za neobičajne dogodke, kot so problemi in napake. Zaradi občutljivosti podatkov rezultati o evalvaciji metode niso podani. V [38] so avtorji predstavili ne-vsiljivo in skalabilno rešitev za detekcijo odvisnosti med aplikacijami med izvajanjem z analizo sistemskih dnevniških datotek. Naredili so tudi obsežno evalvacijo na realnem, velikem sistemu, kjer so z najboljšo metodo dosegli natančnost med 93% in 96%.

II.II Neposredne metode

Primer neposredne metode je opisan v [33], kjer avtorji iz konfiguracijskih podatkov zgradijo graf odvisnosti med komponentami. Njihov pristop najprej primerja konfiguracijske parametre in predlaga odvisnost, če sta konfiguracijska parametra enaka, ali pa je eden podniz drugega. Nato so odvisnosti razvrščene po verjetnostih z različnimi tehnikami. Evalvacijo so naredili na okolju, sestavljenim iz štirih strežnikov, kjer so dosegli 76% natančnost.

Naloga temelji na sistemu, ki je zmožen odkriti podrobne konfiguracije nad različnimi domenami in tehnološkimi skladi avtomatsko v skoraj realnem času. Predlagana metoda za gradnjo grafa komponentnih odvisnosti je torej neposredna metoda. Zaradi pomanjkanja sistemsko specifičnih podatkov, je rešitev z uporabo neposrednih metod trenutno zelo malo. Prav tako predlagana metoda odkrije tudi dinamične odvisnosti, kar v neposredni metodi [33] ni bilo posebej obravnavano. Poleg tega je evalvacija metode narejena na realnem, velikem naboru podatkov, kar je narejeno le v [38].

Predlagana metoda za odkrivanje odvisnosti je zgrajena iz različnih modulov, kjer vsak modul najde odvisnosti na različnih arhitekturnih ravneh – med gostitelji, med aplikacijami in med komponentami aplikacij. Evalvacija je izvedena za vsak modul oz. raven posebej. V kolikor nam je znano, takšna podrobna evalvacija še ni bila narejena.

III Ogrodje za določanje odvisnosti

Ogrodje za določanje odvisnosti je eden izmed znanstvenih prispevkov te naloge. Najprej se bomo spoznali z definicijami izrazov, ki jih bomo uporabili skozi celotno nalogo, nato pa bomo opisali komponente ogrodja.

Definicije, ki so pomembne za razumevanje arhitekture predlaganega ogrodja, so naslednje.

Definicija III.1. ***Gostitelj*** *H je strežnik ali katerakoli druga naprava, ki komunicira z ostalimi gostitelji po nekem omrežju [2].*

Strežnik je računalnik, ki je povezan v omrežje in nudi programske funkcije, ki jih uporabljajo drugi računalniki [4].

Definicija III.2. ***Konfiguracijski objekt (CI)*** *je katerikola komponenta infrastrukture IT, ki je oz. bo pod nadzorom upravljanja sestave in je zato predmet formalnega vodenja sprememb [4].*

Definicija III.3. ***Komponenta IT*** *je prepoznaven del večjega programa [7]. Komponente IT, ki jih je potrebno upravljati, so konfiguracijski objekti [4].*

Definicija III.4. ***Konfiguracijski parameter*** *je podroben, ne-kompleksen podatek, ki je podmnožica konfiguracijskega objekta, katerega vrednost je podvržena spremembam.*

V nalogi se ukvarjamo z iskanjem odvisnosti tudi iz konfiguracijskih parametrov, katerih vrednosti so podvržene spremembam. Torej so komponente IT, ki jih obravnavamo, upravljane in predmet formalnega vodenja sprememb. Zatorej lahko enačimo komponento IT s konfiguracijskim objektom.

Gostitelj je množica različnih konfiguracijskih objektov, kjer vsak od njih nudi skupen nabor funkcionalnosti. Vsak konfiguracijski objekt pa je lahko sestavljen iz različnih konfiguracijskih objektov, kjer vsak od njih nudi manjši, a skupen nabor funkcionalnosti starša in tako naprej. Takšna rekurzivna struktura se imenuje *drevo konfiguracijskih objektov*, katerega korensko vozlišče je gostitelj.

Konfiguracijski parameter je sestavljen iz konfiguracijskega ključa, ki je identifikator (npr. pot do tega konfiguracijskega parametra) in vrednosti, katera je podvržena spremembam.

Definicija III.5. ***Odvisnost*** *je razmerje, ki obstaja med konfiguracijskima objektoma A in B natanko takrat, kadar konfiguracijski objekt A potrebuje storitev, ki jo izvede konfiguracijski objekt B, za izvanje funkcije konfiguracijskega objekta A. [29]*

Definicija III.6. *Kadar je konfiguracijski objekt A odvisen od konfiguracijskega objekta B (to označimo z* $A \xrightarrow{odvisen\ od} B$*), rečemo, da je A odvisnik in B predhodnik [29].*

Lastnost, ki velja za odvisnosti med več kot dvema konfiguracijskema objektoma, se imenuje *tranzitivna lastnost.*

Definicija III.7. *Če je konfiguracijski objekt A odvisen od konfiguracijskega objekta B, in je konfiguracijski objekt B odvisen od konfiguracijskega objekta C, potem je konfiguracijski objekt A tudi odvisen od konfiguracijskega objekta C zaradi tranzitivne lastnosti.*

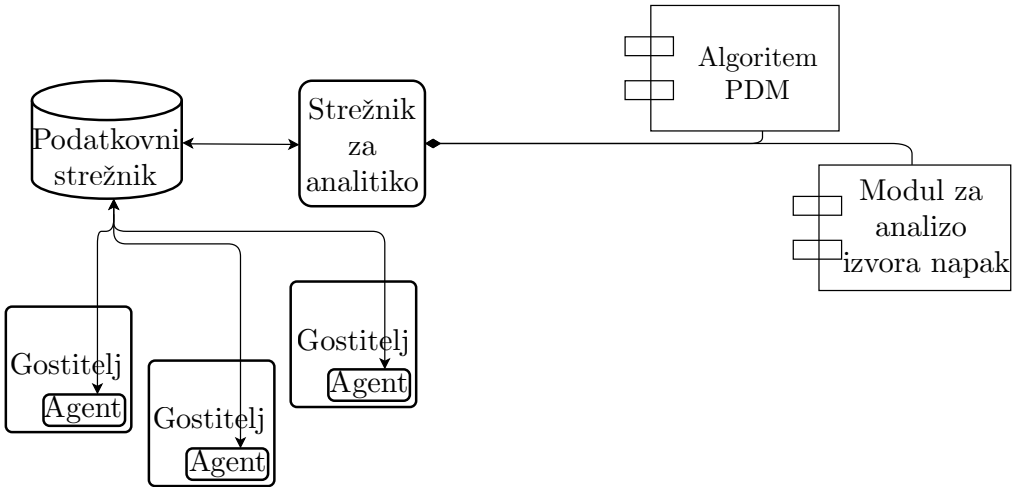
Definicija III.8. *Če je konfiguracijski objekt A odvisen od konfiguracijskega objekta B, potem so vsi otroci konfiguracijskega objekta A (v drevesu konfiguracijskih objektov) in konfiguracijski objekt A odvisni od vseh otrok konfiguracijskega objekta B in konfiguracijskega objekta B.*

Definicija III.9. *Graf, ki vsebuje minimalno število odvisnosti, tako, da se vse ostale odvisnosti lahko dobijo iz tranzitivne lastnosti, ima lastnost tranzitivne redukcije.*

V naši nalogi bomo iskali inter- ter intra-domenske odvisnosti.

Definicija III.10. *Inter-domenska odvisnost je odvisnost med različnimi domenami.*

Vsak gostitelj bo predstavljal svojo domeno, zato so odvisnosti med različnimi gostitelji inter-domenske odvisnosti.



Slika 1: Ogrodje za določanje odvisnosti

Definicija III.11. *Intra-domenska odvisnost je odvisnost med isto domeno, torej so to odvisnosti med različnimi konfiguracijskimi objekti gostitelja.*

Spoznali smo se z vsemi definicijami izrazov, ki jih bomo potrebovali pri ogodju za določanje odvisnosti. Ogrodje, predstavljeno na sliki 1, je sestavljeno iz gostiteljev z nameščenimi agenti, podatkovnim strežnikom ter strežnikom za analitiko, ki je sestavljen iz algoritma PDM ter modula za analizo izvora napak.

Agent je program, ki je nameščen na vseh gostiteljih, na katerih želimo odkriti odvisnosti. Gostitelji so povezani v omrežje, ki omogoča komunikacijo med njimi. Agent periodično pregleda svojega gostitelja in sporoči spremembe, ki so se zgodile v primerjavi s prejšnim pregledom.

Pravila, ki opisujejo rekurzivno strukturo drevesa konfiguracijskih objektov, so opisana v *dokumentu definicij aplikacij*, ki poda tudi navodila, kako razčleniti predefinirane konfiguracijske parametre. Takšen dokument je potrebno definirati za vsako znano tehnologijo. Agent torej sledi navodilom omenjenega dokumenta, z namenom razpoznavanja rekurzivne strukture konfiguracijskih objektov in njihovih konfiguracijskih parametrov, iz katerih je gostitelj sestavljen. Ko agent najde konfiguracijske parametre, jih shrani v podatkovno bazo na podatkovnem strežniku. Konfiguracijski parametri se

lahko shranijo tudi samo takrat, ko pride do spremembe v njihovi vrednosti. [20]

Ko so konfiguracijski parametri shranjeni, lahko začnemo z analizo podatkov in iskanjem odvisnosti med komponentami IT na strežniku za analitiko. Modul za analizo izvora napak strežnika za analitiko poišče ustrezno spremembo, ki je povzročila težavo. Pri tem si pomaga z algoritmom PDM, ki zgradi graf odvisnosti med komponentami, in za vsako odvisnost izračuna verjetnost, da le-ta obstaja. Algoritem je opisan v naslednjem razdelku.

IV Algoritem PDM

V tem razdelku bomo opisali algoritem PDM, ki najde odvisnosti na različnih arhitekturnih ravneh – med gostitelji, konfiguracijskimi objekti ter konfiguracijskimi parametri. Vhod in izhod algoritma so usmerjeni ciklični grafi.

Definicija IV.1. ***Usmerjen graf*** *G je množica vozlišč V in usmerjenih povezav E, označenih z $G = (V, E)$. Vsaka izmed usmerjenih povezav povezuje par vozlišč. [36]*

Definicija IV.2. ***Usmerjen cikličen graf (DCG)*** *je usmerjen graf $G = (V, E)$, ki vsebuje vsaj en usmerjen cikel. Usmerjen cikel v usmerjenem grafu $G = (V, E)$ je usmerjena pot, ki vsebuje vsaj eno povezavo, katere začetno in končno vozlišče sta enaki. Usmerjena pot v usmerjenem grafu G je sekvenca vozlišč, v kateri obstaja usmerjena povezava, ki kaže iz vsakega vozlišča v sekvenci na naslednika v sekvenci. [36]*

V tej nalogi vozlišča V predstavljajo konfiguracijske objekte, povezave E pa predstavljajo odvisnosti med konfiguracijskimi objekti. Če je torej konfiguracijski objekt A odvisen od konfiguracijskega objekta B, dodamo usmerjeno povezavo, ki kaže iz A na B, kot je podano v definiciji III.5.

Vsak gostitelj je s svojo strukturo predstavljen s svojim DCG in je zgrajen v dveh korakih. V prvem koraku naredimo predlogo DCG, ki vsebuje generične odvisnosti med primerki generičnih konfiguracijskih objektov. To

so objekti, ki nudijo podobno funkcionalnost kot so operacijski sistem, spletni strežnik, aplikacijski strežnik ipd. V drugem koraku pa zapolnimo predlogo s podatki gostitelja. Za vsak konfiguracijski objekt najdemo ustrezen generičen konfiguracijski objekt in ga dodamo na to vozlišče. Na primer, operacijski sistem Windows dodamo na operacijski sistem; sistem za upravljanje podatkovnih baz MySQL, ki vsebuje dve shemi, dodamo na vozlišče sistem za upravljanje podatkovnih baz, shemi pa na vozlišče MySQL.

Izhod algoritma je DCG z zgeneriranimi odvisnostmi med konfiguracijskimi objekti, kjer ima vsaka odvisnost dodatno lastnost p, ki predstavlja verjetnost, da takšna odvisnost obstaja.

IV.I Algoritmi

Algoritem PDM je sestavljen iz štirih modulov. Prvi modul, imenovan modul HDE, najde odvisnosti med gostitelji. Drugi modul – modul IDG, najde vse možne odvisnosti med primerki generičnih konfiguracijskih objektov za vse odvisnosti zgenerirane z modulom HDE. Tretji modul – modul CIDE, najde odvisnosti med konfiguracijskimi objekti za vsako odvisnost, zgenerirano z modulom IDG in jim dodeli verjetnost, da takšna odvisnost obstaja. Zadnji modul – modul IDE, dodeli verjetnosti odvisnostim med primerki generičnih konfiguracijskih objektov iz verjetnosti, zgeneriranih z modulom CIDE.

Modul HDE

Modul HDE najde odvisnosti med gostitelji tako, da upošteva naslednjo predpostavko. Če je gostitelj A odvisen od gostitelja B, potem mora gostitelj A vsebovati neko lastnost gostitelja B v konfiguracijskih parametrih. Možno pa je tudi obratno, na primer, če gostitelj A dovoli dostop FTP ali SSH gostitelju B, potem gostitelj A vsebuje naslov IP gostitelja B. Takšne odvisnosti imenujemo obratne odvisnosti. Ena izmed lastnosti, ki razlikuje gostitelje med sabo, je naslov IP. Druga takšna lastnost pa je gostiteljevo ime. Oba podatka lahko dobimo iz agenta ter dokumenta definicij aplikacij.

Modul HDE deluje sledeče. Najprej iteriramo čez vse konfiguracijske parametre vseh gostiteljev in za vsak konfiguracijski parameter preverimo, ali obstaja odvisnost. Če konfiguracijski parameter gostitelja A vsebuje naslov IP ali ime kateregakoli drugega gostitelja, ustvarimo odvisnost z verjetnostjo $p = 1$ med gostiteljema, tako da upoštevamo tudi morebitna pravila za obratne odvisnosti.

Modul IDG

Modul IDG ustvari vse možne odvisnosti med primerki generičnih konfiguracijskih objektov iz odvisnosti, zgeneriranih z modulom HDE, na naslednji način. Za odvisnost med gostiteljem A in B (gostitelj A je odvisen od gostitelja B), ki ni obratna odvisnost, najprej najdemo kateri primerek generičnega konfiguracijskega objekta gostitelja A vsebuje naslov IP ali ime gostitelja B. Nato naredimo odvisnost med najdenim primerkom in vsemi možnimi primerki generičnih konfiguracijskih objektov gostitelja B. Tako dobimo vse možne kandidate za odvisnosti. V primeru obratne odvisnosti med gostiteljema A in B najprej najdemo, kateri primerek generičnega konfiguracijskega objekta gostitelja B ima naslov IP oz. ime gostitelja A v konfiguracijskih parametrih. Nato naredimo odvisnost med operacijskim sistemom gostitelja A in najdenim primerkom gostitelja B.

Tako smo ustvarili vse možne kandidate inter-domenskih odvisnosti. Potrebno je še zgenerirati kandidate za intra-domenske odvisnosti. To naredimo tako, da za vsakega gostitelja zgeneriramo vse možne pare odvisnosti med njegovimi primerki generičnih konfiguracijskih objektov.

Modul CIDE

Ko imamo vse kandidate med primerki generičnih konfiguracijskih objektov, lahko nadaljujemo s kreiranjem odvisnosti med konfiguracijskimi objekti z modulom CIDE. Predpostavke, ki veljajo za odvisnosti med konfiguracijskim objektom A in B, so naslednje. Prvič, oba konfiguracijska objekta A in B vsebujeta ime ali naslov IP gostitelja s konfiguracijskim objektom B.

Drugič, če je konfiguracijski objekt A odvisen od konfiguracijskega objekta B, potem vsebuje objekt A informacijo objekta B v njegovih konfiguracijskih parametrih, kot je to na primer ime konfiguracijskega objekta B (to je lahko ime podatkovne baze, ime aplikacijskega strežnika).

Vendar vsa imena konfiguracijskih parametrov niso primerna, saj med njimi najdemo nekatera zelo pogosta, kot so to privzeta imena podatkovnih baz, aplikacij ipd. Takšne bi radi izločili.

Eden izmed načinov, kako to doseči, je z inverzno frekvenco dokumenta (idf) [27]. Če želimo poračunati oceno idf nekega izraza v korpusu dokumentov, moramo najprej izračunati frekvenco dokumenta $df(t)$ izraza t, ki je definirana kot število dokumentov, ki vsebujejo izraz t. Ocena idf se potem izračuna kot

$$idf(t) = log(\frac{N}{df(t)}),$$

kjer je N število dokumentov v korpusu. Za izraze, ki se redko pojavijo, je ocena idf zelo visoka, medtem ko pa je za zelo pogoste izraze, vrednost nizka.

Da lahko uporabimo oceno idf v naši domeni, moramo najprej definirati, kaj predstavljajo korpus, dokument in izraz v naši domeni. Korpus ustreza specifični tehnologiji, kot je to operacijski sistem Windows, spletni strežnik IIS, podatkovna baza Oracle. Vsak korpus je sestavljena iz dokumentov, ki ustrezajo drevesu konfiguracijskih objektov tehnologije korpusa. Izraz ustreza imenu konfiguracijskega objekta.

Algoritem modula CIDE je predstavljen z algoritmom 1. Za vsako odvisnost, ki je zgenerirana z modulom IDG, najdemo vse tranzitivne odvisnosti s funkcijo $PridobiTranzitivneOdvisnosti(odvisnost)$, kot je definirano v definiciji III.9. Nato dobimo konfiguracijska objekta predhodnika in odvisnika odvisnosti. Če oba konfiguracijska objekta vsebujeta ime ali naslov IP gostitelja predhodnika, si to informacijo shranimo. Z uporabo ocene idf, vsebovanosti informacije o gostitelju predhodnika in z dodatnimi značilkami izračunamo verjetnost odvisnosti s funkcijo *IzračunajVerjetnost(odvisnost)*.

Verjetnost odvisnosti, predstavljene v algoritmu 1 s funkcijo

Algorithm 1 Modul CIDE

Require: $PridobiIzraze(ci)$ ▷ Vrne razčlenjene izraze iz imena konfiguracijskega objekta ci

Require: $PridobiKorpus(tehnologija)$ ▷ Vrne korpus c za specifično tehnologijo $tehnologija$

Require: $PridobiOcenoIdf(t, c)$ ▷ Vrne oceno idf izraza t v korpusu c

for $odv \in odvisnosti$ **do**
  if odv ni $obratna\ odvisnost$ **then**
    $tranzitivneOdvisnosti \leftarrow PridobiTranzitivneOdvisnosti(odv)$
    for $tranzitivnaOdvisnost \in tranzitivneOdvisnosti$ **do**
      $odvisnik \leftarrow tranzitivnaOdvisnost.odvisnik$
      $predhodnik \leftarrow tranzitivnaOdvisnost.predhodnik$
      $izrazi \leftarrow PridobiIzraze(predhodnik)$
      $maxOcenaIdf \leftarrow -1$
      for $izraz \in izrazi$ **do**
        if $izraz \subset odvisnik.CP$ **then** ▷ Preveri, ali se $izraz$ nahaja v konfiguracijskih parametrih odvisnika
          $ocenaIdf \leftarrow PridobiOcenoIdf(izraz, PridobiKorpus(odvisnik.tehnologija))$
          if $ocenaIdf > maxOcenaIdf$ **then**
            $maxOcenaIdf \leftarrow ocenaIdf$
          end if
        end if
      end for
      if $predhodnik.imeGostitelja \subset odvisnik.CP$ **and** $predhodnik.imeGostitelja \subset predhodnik.CP$ **then**
        $tranzitivnaOdvisnost.vsebujeGostitelja \leftarrow True$
      end if
      $tranzitivnaOdvisnost.ocenaIdf \leftarrow maxOcenaIdf$
      $verjetnost \leftarrow Izračunaj Verjetnost(tranzitivnaOdvisnost)$
      $odvisnik \xrightarrow[p=verjetnost]{\text{odvisen od}} predhodnika$
    end for
  end if
end for

IzračunajVerjetnost(odvisnost), izračunamo s pomočjo algoritma za nadzorovano strojno učenje, imenovanega Naivni Bayes. Naivni Bayes je verjetnosti model, ki se nauči pogojnih verjetnosti za vsako izmed n značilk $\mathbf{x} = (x_1, x_2, ..., x_n)$ za vsakega izmed m razredov oz. izidov C_m, $P(C_m|x_1, x_2, ..., x_n)$. Z uporabo predpostavke Naivnega Bayesa, ki pravi, da je vsaka izmed značilk x_i pogojno neodvisna od vseh ostalih značilk x_j, kjer $x_i \neq x_j$ glede na razred C_m, lahko zapišemo enačbo za izračun verjetnosti glede na podan razred kot:

$$P(C_m|\mathbf{x}) = \frac{P(C_m)\prod_{i=1}^{n} P(x_i|C_m)}{\sum_{j=1}^{m}\left(P(C_m)\prod_{i=1}^{n} P(x_i|C_m)\right)}.$$

V našem primeru imamo binaren klasifikacijski problem, saj napovedujemo, ali odvisnost obstaja ali ne. Značilke $\mathbf{x}$, ki jih bomo uporabili so naslednje:

- *domena odvisnosti* nam pove, ali je odvisnost inter- ali intra-domenska;
- *predloga* nam pove, ali odvisnost sledi pravilom predloge;
- *število dokumentov* nam pove, koliko dokumentov se nahaja v korpusu;
- *ocena idf*;
- *vsebuje gostitelja* nam pove, ali se ime oz. naslov IP gostitelja predhodnika pojavi v konfiguracijskih parametrih odvisnika in predhodnika;
- *odvisnost HDE* pove, ali sta oba gostitelja povezana z odvisnostjo, zgenerirano z modulom HDE.

Preden lahko poračunamo $P(C_m|\mathbf{x})$, moramo oceniti pogojne verjetnosti $P(x_i|C_m)$ in verjetnosti razreda $P(C_j)$ na manjši učni podatkovni množici.

Modul IDE

Zadnji modul algoritma PDM, modul IDE, dodeli verjetnosti vsaki odvisnosti, zgenerirani z modulom IDG na naslednji način. Za vsako odvisnost, ki jo naredi modul IDG, pridobi vse tranzitivne odvisnosti med odvisnikom in

predhodnikom te odvisnosti, in najde tisto, ki ima največjo verjetnost. To verjetnost dodeli začetni odvisnosti – odvisnosti med primerki generičnega konfiguracijskega objekta, ki jih je zgeneriral modul IDG.

V Rezultati

Vsakega od modulov, razen modula IDG, smo ovrednotili na podatkovni množici 115 gostiteljev, pridobljeni v velikem mednarodnem podjetju. Celotna podatkovna množica je sestavljena iz več kot 200 primerkov generičnih konfiguracijskih objektov, 5.000 konfiguracijkskih objektov ter več kot 500.000 konfiguracijskih parametrov.

Podatke o gostiteljih ter njihovo strukturo smo shranili na naslednji način. V bazi grafov (angl. *graph database*) smo imeli shranjena drevesa konfiguracijskih objektov gostiteljev kot DCG. V drugi bazi pa smo imeli shranjene konfiguracijske parametre.

V.I Tehnike evalvacije

Ovrednotenje modulov je potekalo na naslednji način. Ne glede na to, kateri modul smo izbrali, smo vedno primerjali napovedane odvisnosti s pravimi odvisnostmi, ki jih je označil strokovnjak. Obstajajo pa različne primerjalne tehnike, ki primerjajo napovedane s pravimi odvisnostmi. V našem delu bomo predstavili tri: osnovno, tranzitivno in tehniko, ki temelji na RCA.

Intra-domenske odvisnosti, ki so nastale zaradi hierarhične strukture dokumenta aplikacijskih definicij, niso vključene v evalvacijo, ker vedno predstavljajo prave odvisnosti.

Osnovna tehnika evalvacije

Mere zmogljivosti, s katerimi bomo evalvirali algoritem, vsebujejo naslednje izraze. Pravilno pozitivna odvisnost (TP) je odvisnost, ki je napovedana kot pravilna in je prav tako označena kot prava. Napačno pozitivna odvisnost

(FP) je odvisnost, ki je napovedana kot pravilna, ampak je označena kot napačna. Napačno negativna odvisnost (FN) je odvisnost, ki je napovedana kot napačna, ampak je označena kot pravilna.

Mere zmogljivosti so naslednje.

Definicija V.1. *Natančnost je razmerje med pravilno prepoznanimi odvisnostmi in vsemi odvisnostmi, ki so napovedane kot prave odvisnosti:*

$$natančnost = \frac{TP}{TP + FP}$$

Definicija V.2. *Priklic je razmerje med pravilno prepoznanimi odvisnostmi in vsemi odvisnostmi, ki so označene kot pravilne:*

$$priklic = \frac{TP}{TP + FN}$$

Mera, ki izračuna harmonično sredino natančnosti in priklica, se imenuje F-mera. Najboljšo vrednost doseže pri 1 in najslabšo pri 0.

$$F\text{-}mera = 2 \cdot \frac{natančnost \cdot priklic}{natančnost + priklic}$$

Pri osnovni tehniki evalvacije seštejemo število TP, FP, FN odvisnosti in izračunamo natančnost, priklic in F-mero. Pri ostalih dveh tehnikah evalvacije pa število TP, FP in FN štejemo drugače.

Tranzitivna tehnika evalvacije

Osnovna tehnika evalvacije ne upošteva tranzitivne lastnosti, ki je definirana v definicijah III.7 in III.8, in pride bolj v poštev pri evalvaciji odvisnosti konfiguracijskih odvisnosti. Evalvacija s tranzitivno tehniko poteka na naslednji način. Najprej zgeneriramo vse tranzitivne odvisnosti za vsako od napovedanih odvisnosti. Vsaka tako pridobljena odvisnost je prav tako napovedana odvisnost. Enak postopek naredimo tudi za realne odvisnosti. Nato seštejemo število TP, FN in FP odvisnosti za pravkar zgenerirane odvisnosti in izračunamo natančnost, priklic in F-mero.

Tehnika evalvacije RCA

Tehnika evalvacije RCA upošteva lastnosti odvisnosti, ki sta bistveni za RCA. Prvič, odvisnost d_i je bolj pomembna, če gre skozi njo veliko število tranzitivnih odvisnosti. Drugič, če je odvisnost d_i edina odvisnost, ki povezuje dve vozlišči, in je pot med njima zelo dolga, mora biti takšna odvisnost bolj pomemba.

Pomembnost odvisnosti d_i lahko izrazimo z utežjo $w(d_i)$, ki upošteva obe zgoraj omenjeni lastnosti. Prva lastnost je povezana z vmesno središčnostjo povezav (angl. *edge betweeness centrality*), ki je definirana kot število najkrajših poti v omrežju, ki gre skozi neko povezavo [21]. Druga lastnost pa nam pove, da naj bo utež odvisnosti $w(d_i)$ sorazmerna največji razdalji, ki obstaja med dvema vozliščema, in gre skozi d_i. Največja razdalja je najdaljša pot, ki obstaja med najkrajšimi potmi med vozlišči. Utež se torej izračuna kot $w(d_i) = \frac{VmesnaSrediščnostPovezav(d_i) \cdot največjaRazdalja(d_i)}{n}$, kjer je n katerokoli število, ki normalizira vrednost uteži na nek razpon. V naši implementaciji smo normalizirali uteži na interval $w(d_i) \in [0, 1]$.

Postopek, ki implementira to tehniko ocenjevanja, je naslednji. Najprej iteriramo skozi vse realne odvisnosti, in za vsako realno odvisnost d_i izračunamo pare vozlišč V_m in V_n, ki gredo skozi d; največjo razdaljo, ki gre skozi pare V_m in V_n; utež odvisnosti d_i; ter inicializiramo prazen seznam napovedanih odvisnosti *no*.

Nato iteriramo skozi vse napovedane odvisnosti, in za vsako izmed njih preverimo, ali je vsebovana v realnih odvisnostih. Če je vsebovana, dodamo napovedano odvisnost v seznam *no*; če pa ni, pa izračunamo njeno utež in dodamo vrednost k FP.

Na koncu sledi še iteracija skozi vse realne odvisnosti. Za vsako izmed realnih odvisnosti dodamo sorazmerno vrednost uteži napovedanih odvisnosti proti parom realnih odvisnosti vrednosti TP in razliko k vrednosti FN.

Iz vrednosti TP, FN in FP lahko izračunamo natančnost, priklic ter F-mero.

Algoritem	Natančnost	Priklic	F-mera
modul HDE	84%	99%	91%
Random 0.02	2%	2%	2%
Random 0.5	2%	52%	5%
EM	27%	93%	42%

Tabela 2: Primerjava rezultatov osnovne tehnike evalvacije za algoritme HDE, *Random 0.02*, *Random 0.5* in *EM*.

V.II Rezultati

V naslednjih razdelkih so predstavljeni rezultati modulov HDE, CIDE in IDE z vsemi tremi tehnikami evalvacije. Rezultati modula IDG niso predstavljeni, saj le zgenerirajo odvisnosti in jim ne dodelijo nikakršnih verjetnosti.

Rezultati modula HDE

Za vse tri tehnike evalvacije smo uporabili enako množico podatkov, v kateri so podatki 115 gostiteljev. Število vseh možnih odvisnosti med gostitelji je $n \cdot (n-1) = 115 \cdot 114 = 13.100$, kjer je n število vseh gostiteljev. Poleg tega smo naredili tudi primerjavo našega algoritma z naslednjimi tremi algoritmi. Prvi algoritem, *Random 0.02*, ustvari naključno odvisnost z verjetnostjo $p = 0.02$, ki predstavlja odstotek vseh realnih odvisnosti. Drugi algoritem, *Random 0.5*, ustvari odvisnost naključno z verjetnostjo $p = 0.5$. Tretji algoritem, *EM*, ustvari odvisnost, če sta oba gostitelja del iste aplikacije.

Rezultati module HDE z osnovno tehniko evalvacije z ostalimi tremi algoritmi so predstavljeni v tabeli 2.

Rezultati kažejo na to, da je naš algoritem uspešnejši od ostalih. Oba naključna algoritma delujeta slabo. Priklic algoritma je EM zelo visok. To pomeni, da je večina odvisnosti med gostitelji znotraj iste aplikacije. Vendar pa je natančnost algoritma EM slabša. Priklic modula HDE je 99%, kar pomeni, da smo odkrili skoraj vse realne odvisnosti. Odvisnosti, ki jih nismo odkrili, so tiste, ki ne vsebujejo naslova IP oz. imena drugega gostitelja za-

Algoritem	Natančnost	Priklic	F-mera
Modul HDE	49%	99%	65%
Random 0.02	2%	51%	4%
Random 0.5	2%	100%	5%
EM	27%	87%	42%

Tabela 3: Primerjava rezultatov tranzitivne tehnike evalvacije algoritmov HDE, *Random 0.02*, *Random 0.5* in *EM*.

Algoritem	Natančnost	Priklic	F-mera
Modul HDE	99.9%	99.9 %	99.9 %
Random 0.02	99.9 %	1 %	2%
Random 0.5	99.9%	55.8 %	72 %
EM	99.9%	99.9 %	99.9%

Tabela 4: Primerjava tehnike evalvacije RCA algoritmov HDE, *Random 0.02*, *Random 0.5* in *EM*.

radi tega, ker manjkajo pravila za razčlenjevanje v dokumentu aplikacijskih odvisnosti. Natančnost modula HDE je slabša, in sicer 84%, zaradi vsebovanosti naslovov IP oz. imen gostiteljev v tabeli gostiteljev operacijskega sistema.

Rezultati tranzitivne tehnike evalvacije so predstavljeni v tabeli 3. Interpretacija rezultatov je podobna kot pri osnovni tehniki evalvacije, le da je v tem primeru natančnost našega algoritma dosti nižja zaradi tega, ker zgeneriramo nove odvisnosti iz odvisnosti FP.

Rezultati tehnike evalvacije RCA so predstavljeni v tabeli 4. Oba naključna algoritma delujeta slabo, v nasprotju pa modul HDE modul in algoritem EM najdeta večino vseh pomembnih odvisnosti.

Rezultati module CIDE

Rezultate modula CIDE bomo evalvirali s tranzitivno tehniko in tehniko evalvacije RCA ter jih predstavili z natančnostjo, priklicem in F-mero. Ker vsebujejo napovedane odvisnosti verjetnosti, da napovedana odvisnost obstaja, rezultate predstavimo sledeče. Za vsako vrednost verjetnostnega praga $t \in [0, 1]$ napovemo odvisnost, če je njena verjetnost večja od verjetnostnega praga t. Nato poročamo natančnost, priklic ter F-mero za vsako vrednost verjetnostnega praga t posebej.

Rezultati tranzitivne tehnike evalvacije modula CIDE so predstavljeni na sliki 2 kot funkcija mer zmogljivosti v odvisnosti od verjetnostnega praga t. Vrednost F-mere je za katerikoli verjetnostni prag manjša od 10%. Vzroki za slabe rezultate so naslednji. Predpostavili smo, da struktura drevesa konfiguracijskih objektov, ki jo dobimo s pomočjo dokumenta aplikacijskih definicij, pravilno opisuje tranzitivno širjenje odvisnosti. Na primer, če je konfiguracijski objekt CI_1 odvisen od konfiguracijskega objekta CI_2, potem so tudi vsi CI, ki so pridobljeni s tranzitivno lastnostjo, del te odvisnosti. Vendar je v nekaterih primerih to le deloma res, saj je lahko starš CI_1 odvisnik. Takšnih odvisnosti, ki jih nismo zgenerirali, je lahko veliko, in le-te tako prispevajo velik delež, ki vodi k slabemu rezultati.

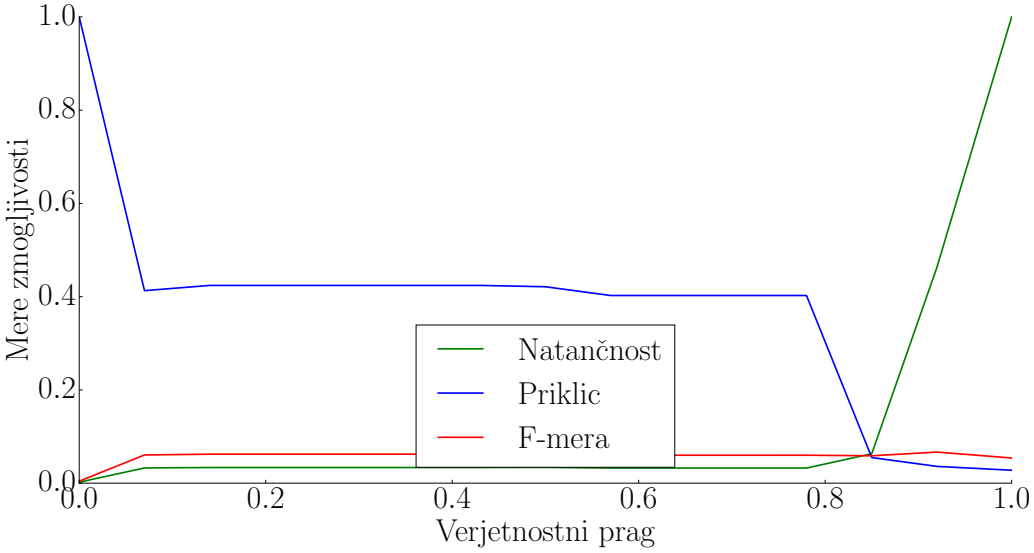
Ta problem je težak in potrebuje dodatno znanje, kako specifična tehnologija deluje in ne more biti rešljiv z našim predlaganim pristopom. Pristop bi lahko razširili z dodatnimi pravili, ki bi podale način zgeneriranja odvisnosti.

Rezultati tehnike evalvacije RCA so predstavljeni na sliki 3 kot funkcija mer zmogljivosti v odvisnosti od verjetnostnega praga t. Vrednost F-mere je za katerikoli verjetnostni prag manjša od 10%. Vzroki za slabe rezultate so enaki kot pri tranzitivni tehniki evalvacije modula CIDE.

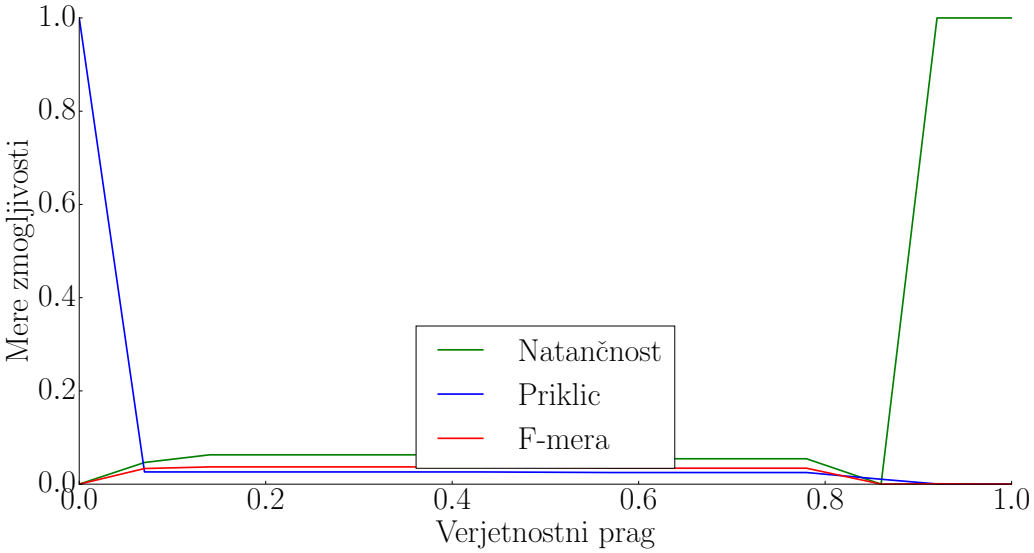
Rezultati modula IDE

Rezultate modula IDE bomo predstavili podobno kot rezultate za modul CIDE.

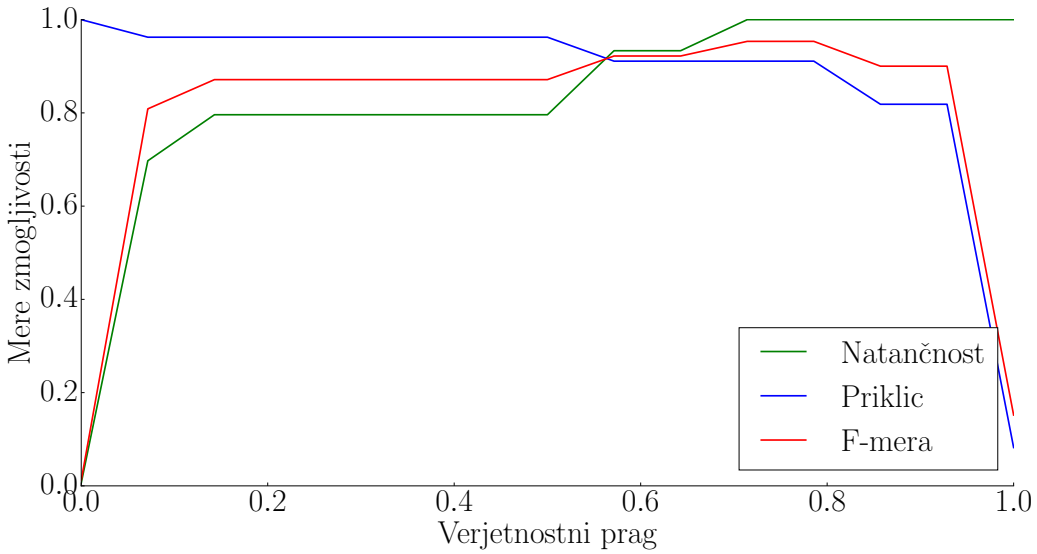
Evalvacija s tranzitivno tehniko je predstavljena na sliki 4 z grafom mer



Slika 2: Graf mer zmogljivosti v odvisnosti od verjetnostnega praga za modul CIDE s tranzitivno tehniko evalvacije.



Slika 3: Graf mer zmogljivosti v odvisnosti od verjetnostnega praga za modul CIDE s tehniko evalvacije RCA.

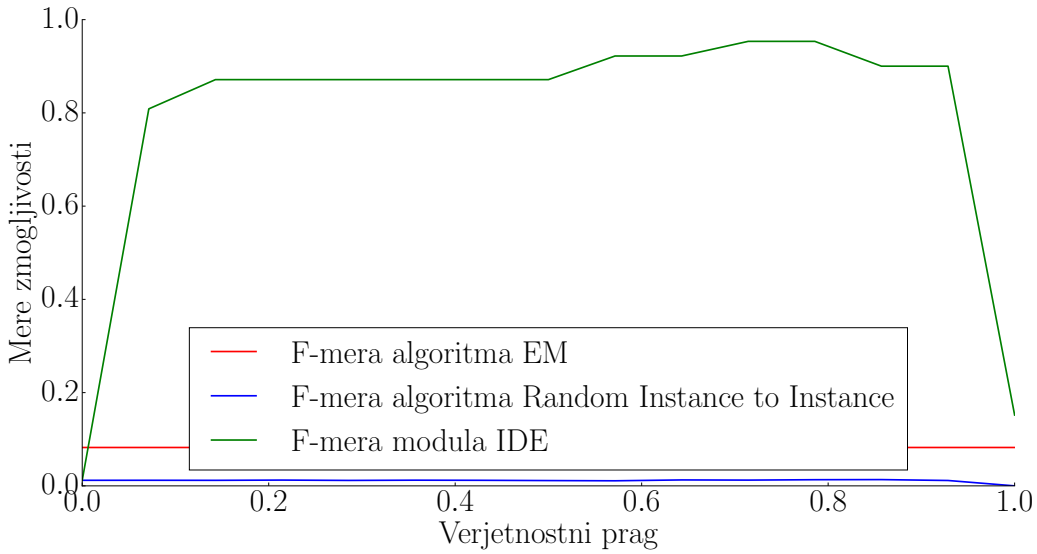


Slika 4: Graf mer zmogljivosti v odvisnosti od verjetnostnega praga za modul IDE s tranzitivno tehniko evalvacije.

zmogljivosti v odvisnosti od verjetnostnega praga. Maksimalno vrednost F-mere 0,95 dosežemo pri verjetnostih med 0,7 in 0,8, kjer je natančnost 1,0 in priklic 0,91. Rezultat nakazuje na to, da so bile skoraj vse odvisnosti napovedane, in vse, ki so bile napovedane, so bile prave odvisnosti. Nenapovedane odvisnosti nastanejo zaradi slabšega razčlenjevanja dokumenta aplikacijskih odvisnosti, neuspešnega pridobivanja konfiguracijskih objektov, oz. so posledica neodkritih odvisnosti v modulu HDE.

Poleg tega smo naredili primerjavo modula IDE z dvema algoritmoma. Prvi algoritem se imenuje *Random Instance to Instance Algorithm*, ki napove odvisnosti med primerki generičnih konfiguracijskih objektov naključno na sledeč način. Za vsak verjetnosti prag t, naključno izberemo število $r \in [0, 1]$. Če je $r > t$, napovemo verjetnost, drugače je pa ne. Drugi algoritem se imenuje *EM* in deluje podobno kot pri modulu HDE. Če sta dva primerka generičnega konfiguracijskega objekta del iste aplikacije, napovemo odvisnost.

Rezultati vseh treh algoritmov so predstavljeni na sliki 5 z grafom F-mere v odvisnosti od verjetnosti. Modul IDE deluje boljše od ostalih, saj odkrije večino odvisnosti z visoko natančnostjo, medtem ko pa imata algoritma *EM*



Slika 5: Graf F-mere v odvisnosti od verjetnostnega praga modula IDE, algoritma *EM* in algoritma *Random Instance to Instance algorithm* za tranzitivno tehniko evalvacije.

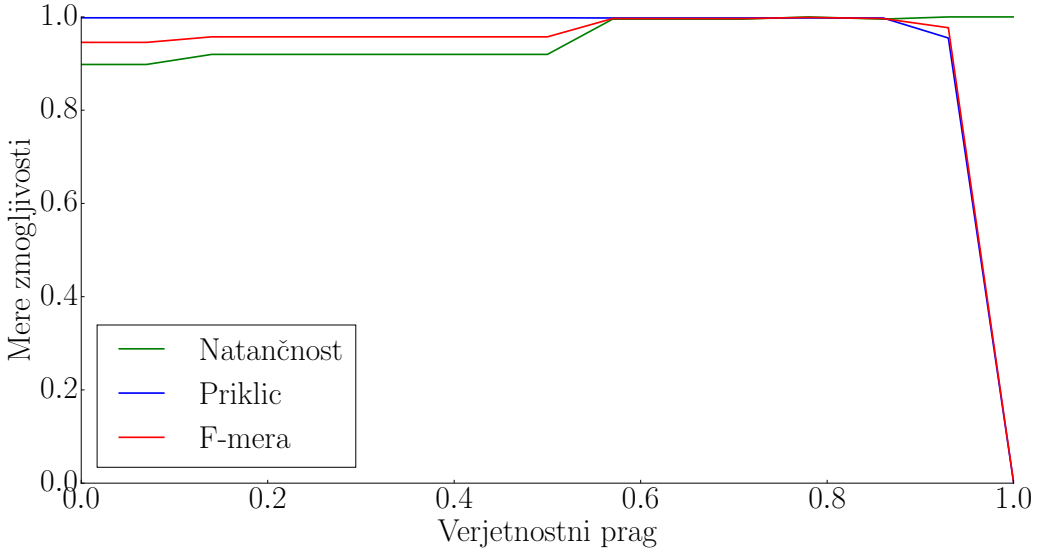
in *Random Instance to Instance* F-mero med 0 in 0,1.

Rezultati modula IDE s tehniko evalvacije RCA so predstavljeni z grafom mer zmogljivosti v odvisnosti od verjetnostnega praga na sliki 6. Rezultati kažejo na to, da pri višjih verjetnostnih pragovih odkrijemo večino pomembnih odvisnosti, saj le-te predstavljajo večji delež v računanju mer zmogljivosti.

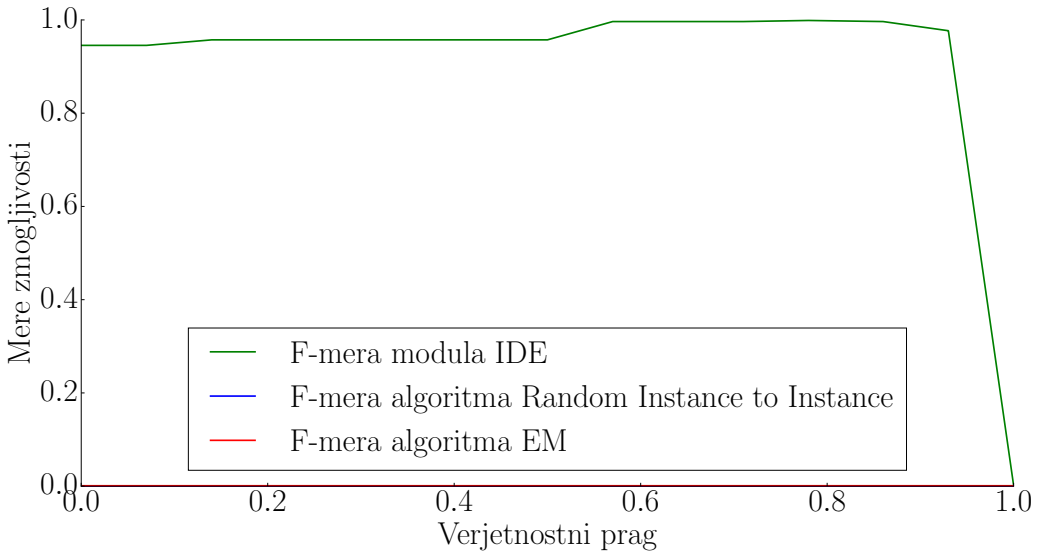
Primerjava modula IDE z algoritmoma *EM* ter *Random Instance to Instance* s tehniko evalvacije RCA je predstavljena na sliki 7. Rezultat kaže na to, da modul IDE deluje zelo dobro napram ostalima dvema, saj imata oba algoritma F-mero zelo nizko – blizu 0.

Učinkovitost zmanjšanja iskalnega prostora

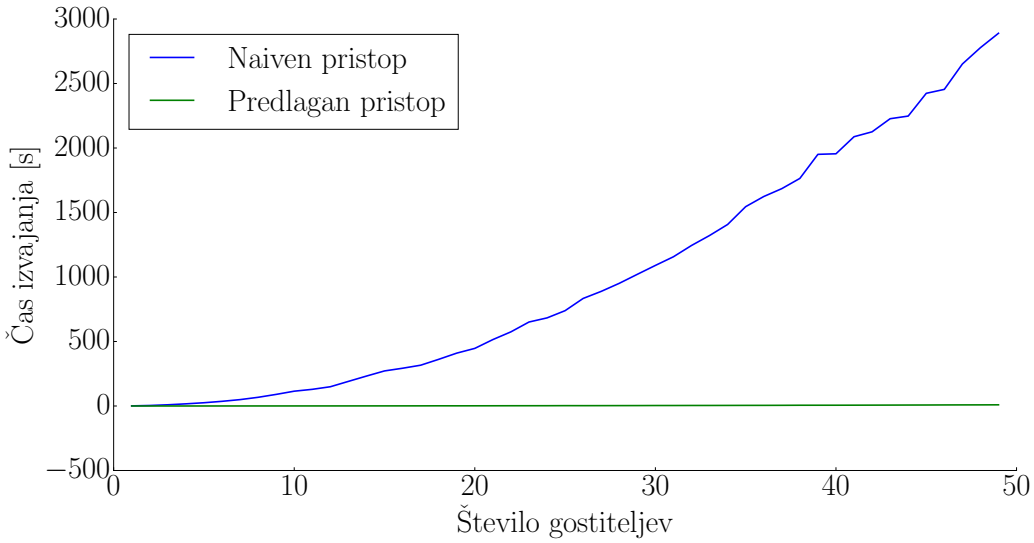
V tem razdelku pokažemo, da z našim pristopom zmanjšamo čas izvajanja in izboljšamo rezultate mer zmogljivost. Namreč, z modulom HDE lahko zelo zmanjšamo preiskovalni prostor, tako da nam v nadaljnih modulih ni



Slika 6: Graf mer zmogljivosti v odvisnosti od verjetnostnega praga za tehniko evalvacije RCA modula IDE.



Slika 7: Graf F-mer v odvisnosti od verjetnostnega praga za tehniko evalvacije RCA modula IDE ter algoritmov *EM* in *Random Instance to Instance*.

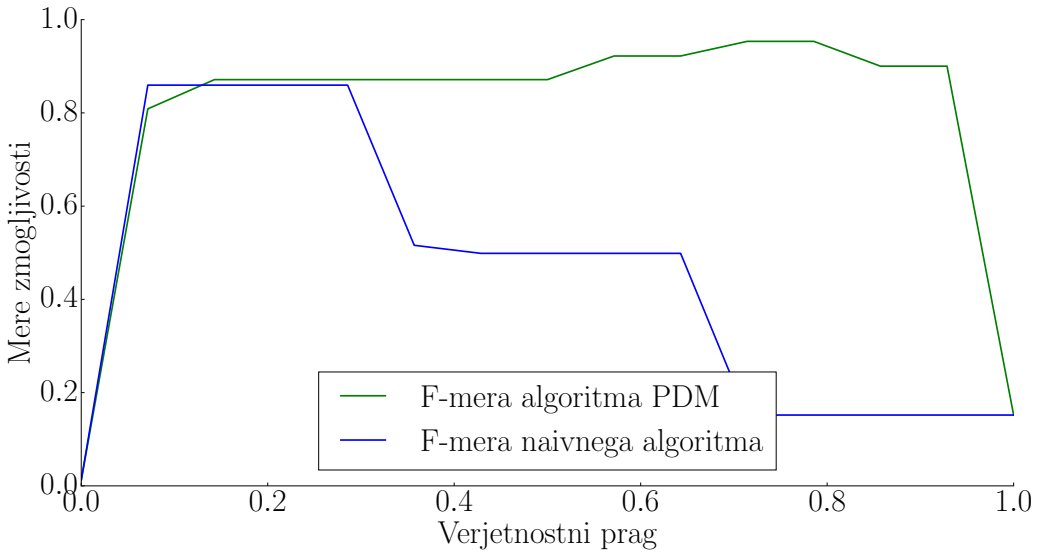


Slika 8: Primerjava časa izvajanja našega in naivnega algoritma v odvisnosti od števila gostiteljev.

potrebno obravnavati vseh konfiguracijskih parametrov.

Naš algoritem smo primerjali z naivnim, ki primerja vsak konfiguracijski parameter z vsakim in se na podlagi tega odloči, ali obstaja odvisnost ali ne. Graf, ki prikazuje čas izvajanja našega in naivnega algoritma v odvisnosti od števila gostiteljev, je predstavljen na sliki 8. Iz grafa lahko razberemo, da je naš algoritem bistveno hitrejši.

Naredili smo še dodaten eksperiment, s katerim demonstriramo učinkovitost našega algoritma v primerjavi z naivnim algoritmom. Lahko predpostavimo, da ne potrebujemo modulov HDE in IDG, ter takoj pričnemo z modulom CIDE, se naučimo pogojnih verjetnosti brez značilke *vsebuje gostitelja*, poračunamo verjetnosti ter jih dodelimo odvisnostim med primerki generičnih konfiguracijskih objektov, kot pri modulu IDE. Rezultate smo prikazali z grafom na sliki 9, ki prikazuje rezultate F-mer modula IDE in naivnega algoritma s tranzitivno tehniko evalvacije v odvisnosti od verjetnostnega praga.



Slika 9: Primerjava F-mer modula IDE in naivnega algoritma, ki primerja vsak konfiguracijski parameter z vsakim, da ugotovi, ali obstaja odvisnost ali ne. Pri tem naiven algoritem ne upošteva rezultatov modula HDE.

VI Sklep

V nalogi smo predstavili neposredno metodo za gradnjo grafa komponentnih odvisnosti, ki iz konfiguracijskih podatkov in strukture gostitelja ugotovi verjetnost odvisnosti med konfiguracijskimi objekti. Predlagana metoda predstavlja naslednje prispevke: formalizacija ogrodja za določanje odvisnosti, ki potrebuje vse potrebne koncepte in komponente, ki jih naš algoritem vsebuje; algoritem, ki z uporabo hevristik, pravil, numeričnih statistik in strojnega učenja ugotovi verjetnost odvisnosti; ter tehnike evalvacije. V kolikor nam je znano, je to edino delo, ki tehnike evalvacije prilagodi lastnostim odvisnosti in izvoru napak.

Rezultate evalvacije lahko le posredno primerjamo z rezultati, ki so pridobljeni iz ostalih sorodnih del, saj so bile evalvacije narejene na različnih podatkih. Ne smemo pa primerjati klasifikacijske točnosti, saj ni primerna za neuravnotežene množice podatkov. V večini primerov je naš algoritem (predvsem modul IDE) boljši. V primerjavi z delom [33], ki primerja konfi-

guracijske parametre vsakega z vsakim, dobimo slabše rezultate z modulom CIDE, vendar pa je avtor evalviral svoj pristop na zelo omejeni množici podatkov.

Predlagana rešitev uporablja neposredno metodo, ki ima naslednje pomankljivosti. Prvič, potrebuje program oz. agenta, ki periodično pridobiva konfiguracijske parametre od gostitelja. Drugič, vsaka tehnologija ima svojo strukturo konfiguracijskih parametrov, zato smo omejeni s kvaliteto in zmogljivostjo pravil, ki le-te razčlenijo. Prav tako je treba ta pravila napisati in jih integrirati s programom, ki pridobiva konfiguracijske parametre, kar je časovno zamudno. Tretjič, omejeni smo s tehnologijami, za katere imamo pravila. Prav tako lahko najdemo le inter- in intra-domenske odvisnosti, ne pa inter- in intra-sistemskih. Le-te lahko najdemo le s posrednimi metodami.

Algoritem PDM ter ogrodje za določanje odvisnosti lahko nadalje uporabimo za detekcijo neznanih gostiteljev, lokalizacijo napak, boljšo analizo izvora napak, ocenitev vpliva nedosegljivosti komponente IT in kot podporo odločitvam o arhitekturi IT.

Chapter 1

Introduction

This chapter introduces the domain and background of the thesis topic, formulates the problem, provides scientific contributions, and gives the overview of the thesis. All the key concepts that we will encounter in this chapter are shown in Figure 1.1 along with the relationships between them.

1.1 Introduction and Background

Large *enterprises* rely on data centers comprising a vast number of servers and associated components, such as networking equipment, storage units, cooling systems, and power supplies [41]. Servers host a plethora of *business applications* that empower business processes, execute business transaction, and serve consumer applications.

Companies aim to have these applications reliable and responsive in order to deliver the services according to a specific Service Level Agreement (*SLA*). SLA is described in Information Technology Infrastructure Library (*ITIL*) [3] as the terms of services that are being offered to the customers [4]. ITIL is a guideline on how to use IT as a tool to facilitate business change, transformation, and growth [5].

SLA is also one of the main aspects of Information Technology Operations (*ITO*). ITO is responsible for the continouos functioning of the infrastruc-

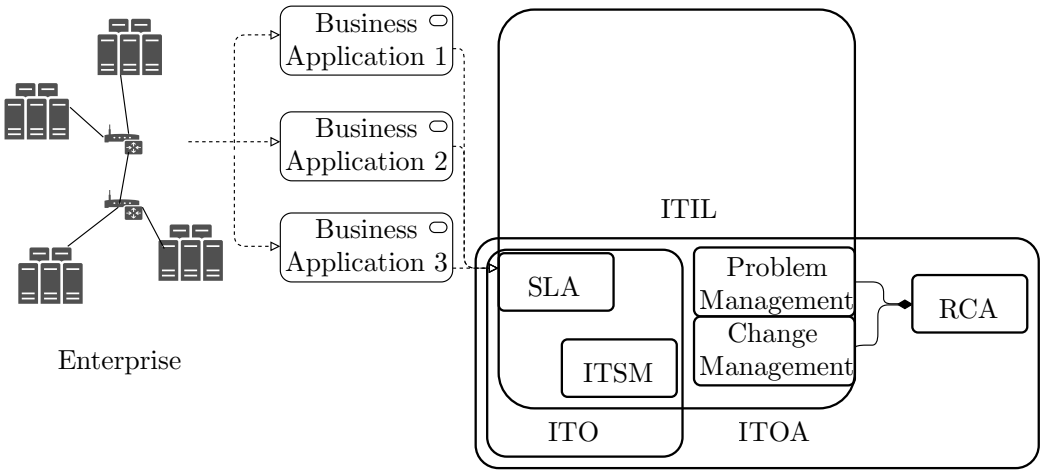


Figure 1.1: Relationships between important concepts mentioned in this section.

ture and operational environments that support application deployment to internal and external customers, including the network infrastructure, server and device management, computer operations, ITIL management, and help desk services for an organization [6].

1.1.1 IT Service Management

ITO processes are associated with ITIL's IT Service Management (*ITSM*), which refers to the set of activities that are performed by an organization to plan, design, deliver, operate, and control information technology (IT) services offered to customers [4]. Therefore goals of ITO and ITSM are to deliver the right set of services at the right quality and at competitive costs for customers [25].

However, problems or incidents do arise and in more than 85% cases can be tracked back to changes applied in IT environments as the industry report indicates [26]. Therefore, the management of problems and changes is very important for resolving the incidents. Both disciplines are described in ITIL as *Change Management* and *Problem Management*.

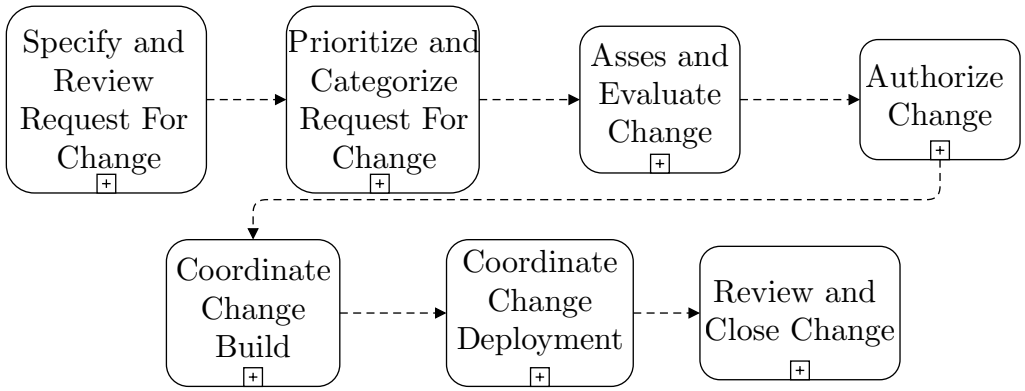


Figure 1.2: Change Management sub-processes

Change Management

Change Management defines how changes should be planned, scheduled, implemented, and evaluated in complex IT infrastructures [16]. The sub-processes that Change Management consists of are shown in Figure 1.2. In the first sub-process, initial specification of a change is specified in a document called Request For Change (RFC) and reviewed. In the second sub-process, RFC is prioritized and categorized by its type, size, and risk. In the third sub-process, the assessment and evaluation of a change is needed in order to establish who should be involved in the assessment and authorization, to asses business justification, impact, cost, benefits, and risk of the change. The change is authorized in the forth sub-process. The levels of authorization depends on the type, risk, and size of a change. In the fifth sub-process, authorized change is passed to relevant technical groups responsible to build the change. In the sixth sub-process, responsible technical group deploy the changes. In the last sub-process, implementation of a change is completed, change is reviewed in order to confirm that the change has met its objectives, and closed.

Even when the organization consistently follows ITIL's practices, problems can occur and are addressed by ITIL's discipline Problem Management.

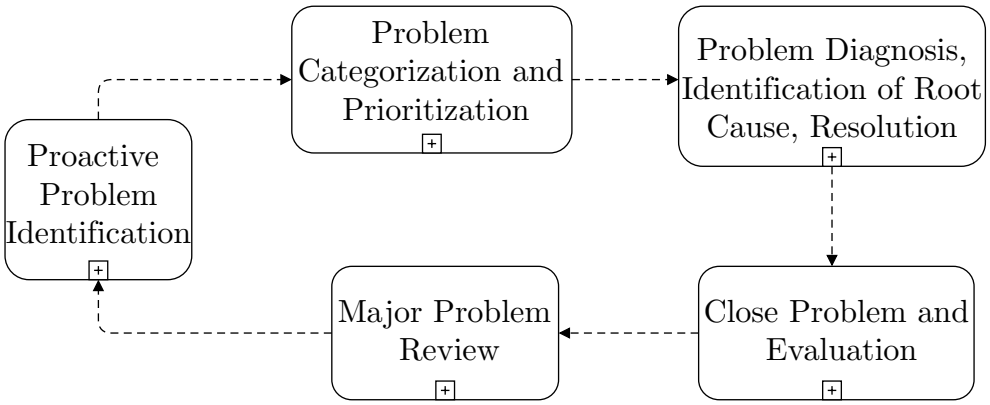


Figure 1.3: Problem management sub-processes

Problem Management

Problem Management describes the management lifecycle of IT problems [3]. Problem Management consists of the following sub-processes that are shown in Figure 1.3. In the first sub-process, the problem is categorized and prioritized in order to promote an effective and quick resolution. In the second sub-process, problem is diagnosed, the underlying root cause of a problem is identified, and resolution is provided. In the third sub-process, problem is closed and evaluated. This is important for future resolutions: if the problem has been successfully resolved, its solution is saved, and can be used for same problems occurring in the future. In the next sub-process, Major Problem Review, the resolution of a problem is reviewed in order to prevent its recurrence and use this information in the future. In addition, the problems that are marked as closed are verified if they have actually been eliminated. The last sub-process is called Proactive Problem Identification. Its aim is to improve the availability of the services by proactively identifying and solving problems.

The main goals of Problem Management are to prevent the occurrence of IT related problems, eliminate recurring problems, and minimizing the impact of the problems on business continuity.

In order to achieve these goals, one has to find the source of a problem.

This can be hard to identify and solve due to the complex structure and size of the IT environment. Since manual discovery and identification of the problems can take significant time and resources, companies are using a practice called IT Operations Analytics (*ITOA*). ITOA is the practice of monitoring systems and gathering, processing, analyzing, and interpreting data from various sources to adopt decisions and predict potential issues [39].

1.1.2 Root Cause Analysis

One of the sub-fields in ITOA is root cause analysis (*RCA*), which covers the process of identifying the source causing an issue in IT environment. For example, changing the firewall settings can lead to service unavailability to users outside the local network. Manual RCA performed by human operators requires expert knowledge of a domain and years of experience, hence ITOA has been extensively focusing on automating RCA [34, 10, 32, 10, 22, 28, 24]. As aforementioned, more than 85% of incidents can be tracked to changes in IT environments [26], which drives the RCA development focus on changes.

In general, RCA system, as shown in Figure 1.4, comprises the following modules:

- Sensing module indicating a SLA violation or other issue with an IT environment, also known as Application Performance Monitoring (APM)
- Component discovery module identifying a list of components in IT system and dependencies between components
- Change history module listing all the changes on particular artifacts or components, also known as Change Management Database (CMDB)
- Reasoning module correlating the reported incident with the most likely root causes

For example, sensing component raises an alert that a purchase transaction on a e-commerce web service takes too long, that is, violates SLA.

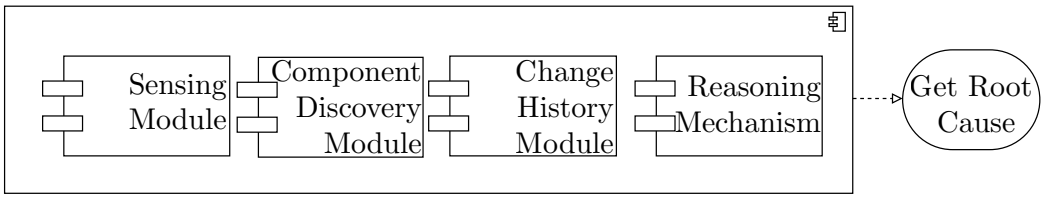


Figure 1.4: RCA system consists of the following components: sensing module, component discovery module, change history module, and reasoning mechanism, which uses information from all previous components to report the most likely root cause that corresponds to the reported incident.

Reasoning component first correlates the alert with a web application hosting that service and tracks all the components the web application depends on, for example, a web server, a database, queues, network routers, etc. Next, the reasoning component queries all the relevant changes on these components and ranks the changes by likelihood that a particular change is the root cause.

1.2 Problem Formulation

In the last decade, the industry developed reliable tools for sensing module and change history module, while component discovery module able to extract components with their granular configurations and dependencies, remains a hard challenge due to the following reasons.

The volume of configuration data and components large enterprise comprises is vast. Even larger is the number of all possible combinations for dependencies between components, which is $n(n-1)$, where n is the number of all components. In addition, configuration data can be changed very frequently. Manually finding and maintaining dependencies between components is not really applicable, since it requires an expert knowledge about components and their dependencies, and it takes a large of time to find the right ones, both of which the operators responsible for RCA are usually lack-

ing of.

Hence, an automatic solution that takes the configuration data to find dependencies among components and responds to dynamic changes is needed in order to build a component dependency graph [34], which is the key element in RCA providing information how the components and sub-components are dependent on each other.

1.3 Scientific Contributions

This thesis provides the following three scientific contributions. First, we formalize the dependency mapping framework, in which we present all the essential components needed in order to build a component dependency graph.

Secondly, we implement the algorithm, which uses granular configuration data, and the structure of a host in order to determine the probability of a dependency between each IT component. It comprises three main modules, where the first module efficiently reduces the search space for configuration parameters. The last module uses a combination of supervised learning method and numerical statistic in order to determine the probability of a dependency.

The last scientific contribution are the evaluation techniques suitable for the component dependency graph. We adjust the evaluation to include the transitive property of a dependency, and we also provide a root cause based evaluation technique. Both evaluation techniques has not yet been considered to the best of our knowledge in the component dependency discovery.

1.4 Overview of the Thesis Structure

The rest of the thesis is structured as follows. In Chapter 2 we provide an extensive literature review of the existing approaches on building a component dependency graph. In Chapter 3 we define a framework that is needed to create the component dependency graph. In Chapter 4 we describe the

algorithm for dependency discovery. Experimental setup with implementation details and evaluation results are described in Chapter 5. We conclude the thesis with the summary of its main contribution and limitations of our approach in Chapter 6, where we also give directions on the future work.

Chapter 2

Related Work

In this chapter we give an overview of the related work along with different methods that are used for constructing a component dependency graph; we review both, the indirect and direct methods within. We conclude this chapter by summarizing the key similarities and differences between our and related work.

2.1 Introduction

Component dependency graph can give us information about the dependencies among components and their sub components, which is important for determining the cause of problems that have been propagating throughout the system [41] – RCA. The methods for constructing a component dependency graph are shown in Figure 2.1.

There are two methods of constructing a component dependency graph: indirect and direct methods [23]. Direct methods need human or static analysis program to analyze the system configuration, installation data, and application code in order to compute dependencies and are system specific.

Indirect methods operate at run-time. They are further divided into intrusive, semi-intrusive, and non-intrusive methods, depending on the level of dependence on code instrumentation. Intrusive method rely strictly on code

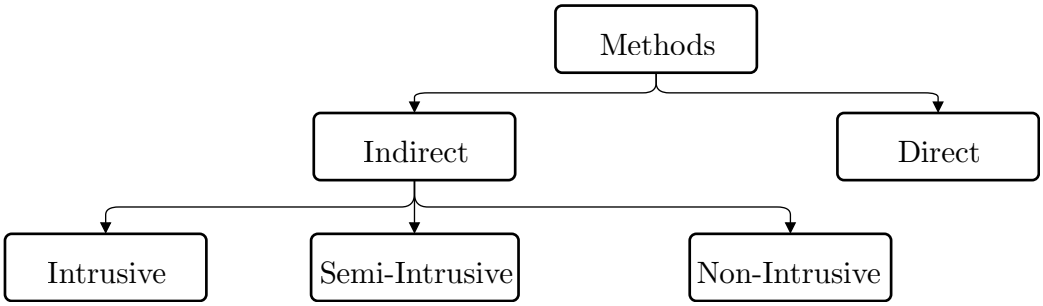


Figure 2.1: There are two main methods for constructing a component dependency graph: direct and indirect method. Indirect methods can be further divided into intrusive, semi-intrusive, and non-intrusive methods.

instrumentation, where as non-intrusive methods do not. Intrusive methods are not suitable in large data centers due to the following reasons. Code cannot be inserted in the system due to the security or licensing issues, such methods cannot be used in an environment where components are coming from different vendors. Therefore semi-intrusive, non-intrusive, and direct methods are more suitable.

The key differences between indirect and direct method are the following. Direct methods requires a knowledge about the system, whereas indirect methods do not need any specific knowledge. Therefore, direct methods are system specific in comparison with indirect methods, which are more general and can work on different systems. Both methods might not be able to discover all the dependencies. Direct methods are not able to discover dependencies between components that are non-specific to system, whereas indirect methods are not able to discover dependencies between components that do not operate at run-time.

There are many different ways of constructing the component dependency graph, which can strictly follow one method or can be a mixture of different methods. Industry products are also already available. Summary of the related work is presented in the Table 2.1 along with the experimental setup and evaluation.

In the following sections we provide a review of indirect and direct meth-

Table 2.1: Summary table of the related work

Reference	Methods	Industry product	Experimental setup	Evaluation [a]	Dependency type
Chen et al. [14]	intrusive indirect	no	test application, 133 test cases	provided results only for fault detection	between applications' components
Brown et al. [13]	semi-intrusive indirect	no	small test application, 11 test cases	A: 99%, P: 97%	between applications
Chen et al. [15]	semi-intrusive indirect	no	5 Microsoft application in Microsoft network	A: 99%, P: 21%	between applications
Aguilera et al. [11]	semi-intrusive indirect	no	test application, with 21 clients	false positive: 21-29%	between applications' components
ServiceNow [37]	direct, semi-intrusive	yes, ServiceNow	n/a	n/a	between hosts, applications' components
VNT Software [40]	direct, semi-intrusive	yes, IlumniIt	n/a	n/a	between hosts, applications
Gupta et al. [23]	non-intrusive indirect	no	3 servers	A: 100%, P: 63-100%	between applications' components
Ensel [19]	non-intrusive indirect	no	n/a, only architecture described	n/a	between hosts, applications
Marvasti et al. [32]	non-intrusive indirect	no	online banking companies	correctly identified causes	between applications
Steinle et al. [38]	non-intrusive indirect	no	large, real clinical system	best P: 93% - 96%	between applications
Ramachandran et al. [33]	direct	no	application, database, portal, messaging server	P: 76%	between configurations

[a] We list one of the performance measures that were mentioned in an article. The abbreviations stand for the following: A is the accuracy and P is precision. In the cases where the total average measure wasn't provided, we calculated it. Performance measure is stated in the range, when it depends on the experimental settings, algorithm parameters, etc.

ods with their related work.

2.2 Indirect Methods

In the following sections we provide a summary of related work that constructs the component dependency graph with intrusive, semi-intrusive, and non-intrusive indirect methods.

2.2.1 Intrusive Method

An example of an intrusive indirect method is presented by Chen et al. [14]. The authors developed a system called PinPoint that dynamically discovers dependencies between applications' components in the following way. The client requests are marked and traced as they are traveling throughout the system through different components and in the same time, PinPoint is discovering components. The drawback of this work is that it doesn't describe the dependency graph more into the details and it does not give evaluation results of the dependency graph due to the focus on fault detection. Nevertheless, the authors simulated the users, fault injection, and provided the experimental results in terms of accuracy and precision.

2.2.2 Semi-intrusive Method

Approaches, which are using semi-intrusive indirect methods, are presented by Brown et al. [13], Chen et al. [15], Aguilera et al. [11].

Brown et al. [13] developed a technique, called Active Dependency Discovery, which discovers dependencies between applications via fault injection and perturbation. The procedure for building a component dependency graph consists of four steps. In the first step, it identifies hardware and software components relevant to the failure. This information is obtained from an outside source. Instrumentation of the system is performed in the second step. In the third step, active perturbation is applied to the system in order

to discover dependencies in the following way. Workload is applied to the system and at the same time, components are perturbed at different levels of intensity. The system behavior, performance and availability is recorded in between. In the forth step, analysis of perturbation data is performed which unveils the dependencies between components.

The validation of this technique was performed on a small, fully functional web-based e-commerce environment, consisting of three tiers. They performed 11 experiments. The number of all potential dependencies is 140, 42 of them were true dependencies. Their technique discovered dependencies with high accuracy and precision: 99% accuracy and 97% precision. However, this is only a small experiment and it would need validation on a much bigger, real environment.

Chen et al. [15] developed a tool, called Orion, which finds dependencies between services and applications by analyzing application traffic. It uses information from the packet headers (IP, UDP and TCP) and timing information in order to discover dependencies. It consists of three components. In the first one, it converts network traffic traces into flows in order to infer the boundaries of application request or reply. In the second component, it first identifies potential services from the flows and then, it computes delay distributions between flows of different services. In the last component, it filters noise and discovers dependencies based on the delay distributions. The underlying assumption is that if a service A depends on a service B, delay distribution should not be random.

Orion focuses on discovering dependencies between hosts' services, which also makes it scalable. One downside is that it requires a large number of samples in order to extract dependencies with high performance measures. This means that infrequently used and new services are not correctly extracted. The authors evaluated Orion on Microsoft's corporate network which consists of large number of application, however the experiments were carried out only on five of them. The authors monitored the traffic on three routers for a two week period. There were 9 local area networks at the first router,

with 2048 clients. The second and third router consisted of two servers in the data center. Orion performed with 99% accuracy and 21% precision over five preselected applications.

Aguilera et al. [11] developed an approach, which relies on tracing the application requests and responses between different applications' components and using one of the algorithms to detect causalities from these traces. It consists of three steps. In the first step, they obtain complete trace of all inter component requests and responses for an operating system. The seconds step consists of post-processing the trace using one of the algorithms, which gives an output of detected dependencies between application components. In the third step the authors provided visualization of the results.

They evaluated their algorithm using an example web-based application. A load generator was also on the same hosts, emulating 24 clients. Their algorithm resulted in false positive rate (dependency was predicted but it was not a true one) between 21%-29%

Building a component dependency graph using a network traffic has also been successful in the industry with the products such as ServiceNow [37] and IlumniIT [40], which are a mixture of semi-intrusive indirect and direct methods since they also rely on configuration data. Both approaches discovers dependencies between hosts and applications.

2.2.3 Non-intrusive Method

Non-intrusive indirect methods either rely on obtaining performance data from the system, since vendors usually already provide simple performance metrics [23], are obtaining this data using different tools [19, 32] or are mining logs in order to discover the dependencies [38]. The idea is that two components are dependent if their performance activity is happening at approximately same after after observing such a behavior several times.

Gupta et al. [23] build the system which discovers dependencies between applications' components from the run-time data. Their experiment consists of three servers. The first server is a test environment and consists of web,

application server, and database. The next one simulates real users using the test environment by sending URL requests. The last one analyses and builds the component dependency graph from the run-time data obtained from application server and database. Note that they only considered synchronous requests. Dependencies were discovered with 100% accuracy and precision ranging from 63% to 100%, depending on the simulation load; however their test environment was very small, consisting of only one server.

Ensel [19] is using neural networks, which are fed with time series of workload data in order to build the component dependency graph, which depicts dependencies between hosts and applications. However only architecture and procedure is described without any experiments and their results. Another drawback of this method is that neural networks are supervised technique and therefore, they require labeled dataset for training, which is not always available. Nevertheless, this approach can be applied to numerous different settings due to the usage of very general data (such as CPU load, TCP/IP communication).

Marvasti et al. [32] are building probabilistic directed graph using workload data for abnormal events (problems or faults). Nodes represents events and the connection between them are conditional probabilities of each pair. They also build a root cause analyzer upon the dependency graph. Similarly to Gupta et al. [23], asynchronous requests are not considered. Another drawback of this method is that it can produce plenty of false dependencies, especially in the event of a high volume of simultaneous requests. Experiments were done on the real dataset of several banking companies with very complex infrastructure to discover the dependencies between applications. Due to the sensitivity of the information, the results in terms of evaluation measures were not provided, however the testing was successful.

Agarwal and Madduri [10] solved the problem of asynchronous requests, which has not been dealt with in [32, 23], however they assume that component dependency graph is already known.

Steinle et al. [38] presented non-intrusive and scalable solution for detecting dependencies between applications at run-time with mining system logs. Their solution was developed for the clinical system for the Geneva University Hospitals, where the availability of a system is crucial. The authors developed three different methods for discovering dependencies. In the first method, one can see the logs as simple activity statements at given time. The technique is based on temporal mining from event logs [31] and is very general. In the second method, logs are seen as a simple activity within the context of user session. In order to identify to which session log corresponds to, some structure or external information is needed. In this way, the logs can be identified that stems from the same user session. This method tries to minimize the parallelism of the simultaneous user's activity that exists in the first method. Note that parallelism due asynchronous requests still exists. The last method is based on analysis of the free text. Invocation of the service is usually logged by application developer, however there is no standardized way of creating such logs and need to be analyzed as free text.

Steinle et al. [38] also provided an extensive evaluation on their real system for each of the methods. The authors used the log data for one week, which corresponds to 56.8 million logs. A reference model for each of the methods was also provided. For the first and second method, the model consists of pairs of application logs, which are dependent if they are directly interacting. For the third method, model is a set of pairs created by an application or logs and a service directory entry that this application is using. In the first model, used by the first and second method, there are 54 applications, which results in 1431 different pairs, 178 of them are true dependencies. In the second model, used by the third method, there are 52 application, 47 service directory entries, and 177 true dependencies. It is important to note that these reference models are static, whereas the approaches are dynamic. This means that there can exist dependencies which rarely occur and will not be captured with their dynamic models if they do not happen during log mining.

The results of each of the methods are as follows. Note that precision has been calculated from the article. With the first method, precision between 68% and 73% has been reached. The second method was a bit better. On the weekdays it discovered dependencies with 75% precision, and on weekends 72%. The third method performed best; precision was between 93% and 95% for the weekdays and 96% for the weekends. Note that the third method is performing better on weekends due to the lower amount of users using the system and, consequently, lower parallelism.

This is the only article that tested their solution on a real system, providing comprehensive evaluation details. Their best method is only suitable for their dataset.

2.3 Direct Methods

An example of a direct method is presented by Ramachandran et al. [33], who are using configuration data to construct a component dependency graph. This approach first compares configuration parameter values across different environments and suggests a dependency if values are equal or a value is a substring of another. Secondly, dependencies are ranked using various techniques. One of the drawbacks of using direct method is that they are system specific and need plenty of information about the system. Obtaining such information requires a lot of effort.

The authors evaluated their method with two case studies. In the first study, the test environment consists of two servers: application and database server. In the second, they have added portal and messaging server. They reached 100% precision for the first case study while using the best ranking technique – this ranking technique sorted the dependencies in such a way that all of the true dependencies were at the top. For the second case study, in which there are 43 true dependencies (in comparison with the first one, where there are only five of them), the precision with the best ranking technique reached 76%. Additional case study, which would include real data from large

enterprises, would gives better estimation of the suitability of this method to an enterprise environment.

Another way to express general dependencies – dependencies that always exist, for example, a dependency between a database and application server, is with an abstract model [19]. One of the drawback is that it has to be manually build and maintained as technologies are changing, however this model is very general and can be later used with any of the approaches described above in order to infer the exact dependencies between components.

2.4 Summary

There are many different methods and approaches to build a component dependency graph; however, not all of them are based on the same assumptions. Most of the indirect methods are able to estimate dependecies between active components only, that is, if a component is inactive, no instrumentation data will be available. The main limitation of the direct methods is that it requires either an expert that manually analyzes relevant configurations or an extensive data collection infrastructure as demonstrated on a limited domain by Ramachandran et al. [33].

This thesis is build upon a recent commercially-available system that is able to detect granular configurations accross multiple domains and technology stacks automatically in near real time [1]. Such detailed and granular configuration data was not available to prior research.

Our method for building a component dependency graph is a direct method, such as is proposed by Ramachandran et al. [33]. Due to the lack of availability of a system specific data, there are currently very few solutions for building a component dependency graph using a direct method. Our method takes into account the dynamic nature of dependencies, as is mostly considered in indirect methods ([32, 14, 38, 23, 15, 13]). However, compared to the direct method in [33], dynamic dependencies were not specifically addressed. In addition, compared to the most of the related work, the evaluation of our

method is done on the real, big dataset, not on a small, test environment. The extensive evaluation on a real dataset has only been done by Steinle et al. [38], however the authors used indirect method.

One of the scientific contributions of this thesis is a novel automated direct method for constructing component dependency graph from granular configuration data. The proposed method is combined of different modules, where each module finds dependencies at different architectural level – between hosts, applications, and applications' components. Additionally, each module assigns the likelihood that the dependency exists. The evaluation is performed for each module and consequently, for each architectural level, separately. Such evaluation has not yet been done to the best of our knowledge.

Chapter 3

Dependency Mapping Framework

In this chapter we formalize the dependency matching framework, the first scientific contribution of this thesis. First, we give a motivating example why discovering dependencies between different IT components is necessary. Secondly, we define the terms that are essential for describing the proposed dependency mapping framework. Finally, we describe all the components comprising the proposed dependency mapping framework.

3.1 Introduction

Mapped dependencies between IT components are an important feature in many IT challenges. The example in the following section illustrates their importance.

3.1.1 Motivating Example

Consider the following example. A company is developing a system for small hotels, which allows booking a hotel room, overview of the availability of hotel rooms, analysis about guests and their stay, and guest check-in upon arrival. The architecture consists of three servers: front-end, middle-ware,

and back-end server. Front-end server has a web server installed, middleware server has an application server installed, and back-end server has a relational database. Described architecture is shown in Figure 3.1. Web server serves web content over HTTP protocol to the end users (either a potential guest or a hotel staff) via browser and gets the actual content (such as guest information, occupancy of the rooms, statistics, booking a room etc.) from the application server. Application server executes business logic: creates new booking, calculates statistics, adjusts the prices according to the season and availability of the hotel. In order to get the data for this tasks, it connects to the relational database on the back-end server. On this server, the actual data about guests, bookings, and room availability is stored.

A component dependency graph for this example is quite simple, as shown in Figure 3.1. The web server depends on the application server and the underlying operating system; the application server depends on the relational database and the underlying operating system; and relational database depends on the underlying operating system.

Such component dependency graph can help us finding the root cause of an issue. For example, assuming someone has changed a configuration in the database, which in turn caused problems in the application server.

Finding the root cause of this problem using component dependency graph proceeds as follows. First, we check the application server and its operating system to see if there were any new deployments or changes in the past week that could cause the problem, however nothing as such has occurred. Therefore, we check the first component that our server depends on, the relational database. We perform the same actions there and see that something has changed, which is causing the initial problem and we quickly resolve it. We did not have to check the web server, because the root cause of the problem would not be there. Therefore, we saved some time with the component dependency graph by not checking the web server. However, this is a very simplistic example on a small-sized architecture, which does not

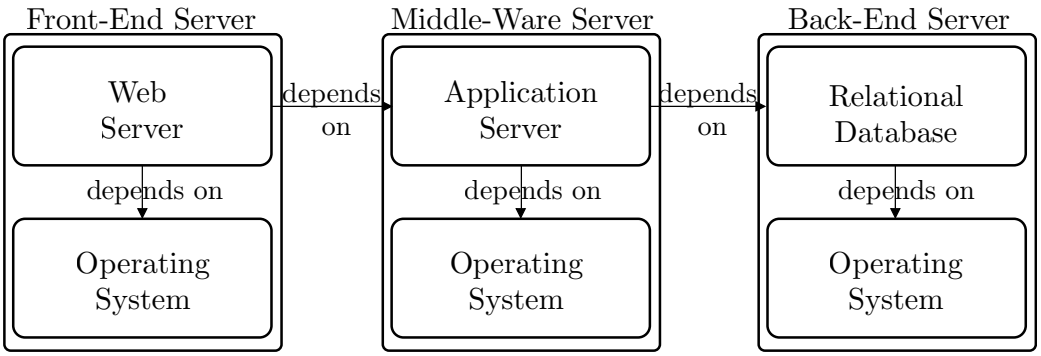


Figure 3.1: Architecture of the small hotel system, consisting of three servers. The front-end server consist of web server, which serves the web content. It gets the content from the middle-ware server, on which it depends to. The middle-ware server consists of an application server, which executes business logic. The data comes from the back-end server. Therefore the application server depends on the back-end server. The back-end server consist of the relational database, where the data is stored.

show us the real power of component dependency graph. Therefore, consider the following.

Imagine now that this company is gaining a larger market and suddenly attracting larger hotels. The architecture for our application suddenly becomes more complex, consisting of more than 50 servers and network devices to scale the traffic efficiently and provide high availability of the system. For illustration, one such environment is depicted in Figure 3.2. A load balancer distributes the traffic across the front-end servers. The general, expected dependencies are illustrated with arrows and are the following. A front-end server depends to any of the application middle-ware servers. An application middle-ware server depends to message queue middle-ware servers and a message queue middle-ware server depends to an application middle-ware servers. An application middle-ware server also depends to back-end servers.

To monitor the SLA violations and other issues with such a complex infrastructure, the usage of APM tools is necessary. This tools create an alert

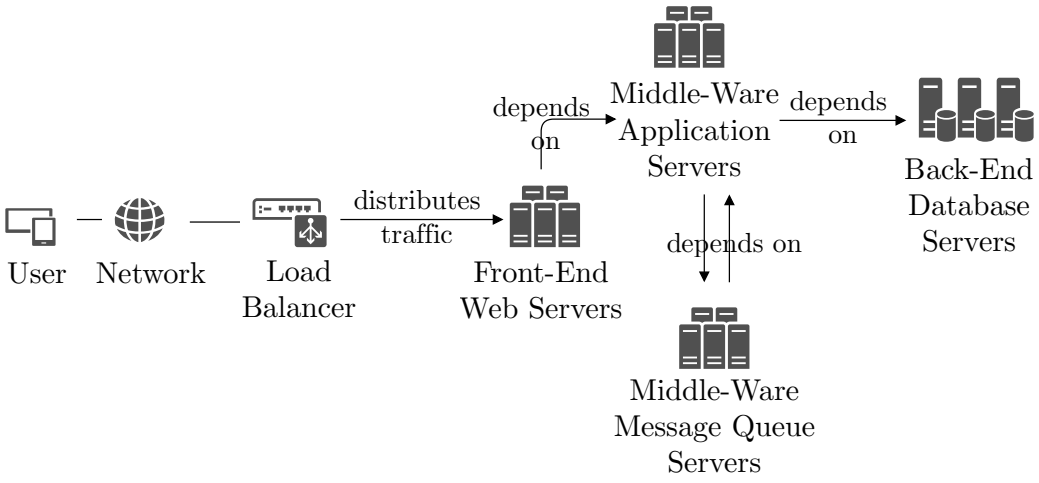


Figure 3.2: The complex architecture of the test environment of our system. The arrows shows general, expected dependencies between groups of servers. Note that now, constructing a component dependency graph is much harder, since we do not know to which of the servers (and their components) one server depends to.

when something is out of ordinary, for example, critical business transaction is taking too long. With RCA system build upon this infrastructure, finding the root cause can be significantly faster. First, RCA system will correlate the APM alert with the server this alert points to – involved server. Next, it will identify all the components this host or application depends on. After, changes that happened before this alert on all the dependent components and on involved server are obtained, correlated with the alert, and prioritized with the probability of causing the alert.

In this thesis we are focusing on building a component dependency graph to find out to which components the affected server depends on. In small environments this is not really necessary as we have seen from the first example. However, when an architecture becomes increasingly more complex, as shown Figure 3.2, it becomes much harder to build it and maintain it. Manually building dependency component graph of such complex environments requires an extensive knowledge of all of the components, takes a lot of

time, and, in addition, it needs to address dynamic changes, i.e., new servers can be added or old ones can be removed in an instant, new dependencies can appear or the old ones can be removed right away, etc. Therefore, an automatic method that addresses dynamic issues is needed to build a component dependency graph, which helps the RCA to perform better, reduces the mean time to resolution with a minimum down time of the system, and minimizes the loss of revenue.

In this chapter we define a framework that builds a component dependency graph. The algorithms that create the dependencies between components are presented in the Chapter 4.

3.2 Definitions

In this section we define the terms that are used in the rest of the thesis and are important for understanding the architecture of the proposed framework. We define the terms from the largest component to the most granular one, reusing some of the ITIL's definitions.

Definition 3.2.1. Host H is a server or any other device (such as a network device) that communicates with other hosts on a network [2].

A server is by ITIL's definition a computer that is connected to a network and provides software functions that are used by other computers [4]. The entry point to a host is its IP address.

Definition 3.2.2. Configuration item (CI) is any component of an IT Infrastructure, including a documentary item such as a SLA or a RFC, which is (or is to be) under the control of Configuration Management and therefore subject to formal Change Control [4].

Definition 3.2.3. IT component is an identifiable part of a larger program or construction [7]. IT components that needs to be managed should be configuration items [4].

Definition 3.2.4. Configuration parameter is a granular, non-complex item, which is a subset of a configuration item whose value is a subject to change over time.

Our work partly relies on inferring dependencies from configuration parameters, which are subject to change. Therefore, IT components that we consider need to be managed and are subject to Change Control. Consequently, we can equate IT components to configuration items.

Host is a collection of different configuration items. Each of them provides a common set of functionality. Note that each one of them can be further composed of different configuration items, each of them providing smaller set of functionality of the parent and so on. This recursive structure is called *configuration item tree*, where the root node is a host.

The configuration item that we encounter first is the operating system installed on the host. Every application installed on this operating system is its own configuration item, such as database management system, application server, or web server. Every database that database management system manages can be its own configuration item, as well as every different application or web application in application or web server. An example of different configuration items is shown in Figure 3.3.

The most granular configuration items consist of configuration parameters. Such host structure is shown in Figure 3.4.

Configuration parameter consists of the configuration key, which is an identifier (for example, a path to this configuration parameter) and its value, which is subject to change. Configuration parameters can include hardware details (e.g. size of RAM, disk size, available disk space), firmware details (e.g. BIOS version, BIOS size), and software application details (e.g. application pool size, connection string, port number, installed drivers or updates, file names, sizes, and checksums belonging to an application).

We extend the Definition 3.2.3 to define CI component:

Definition 3.2.5. CI component is any IT component, which provides the functionality of the parent component in configuration item tree.

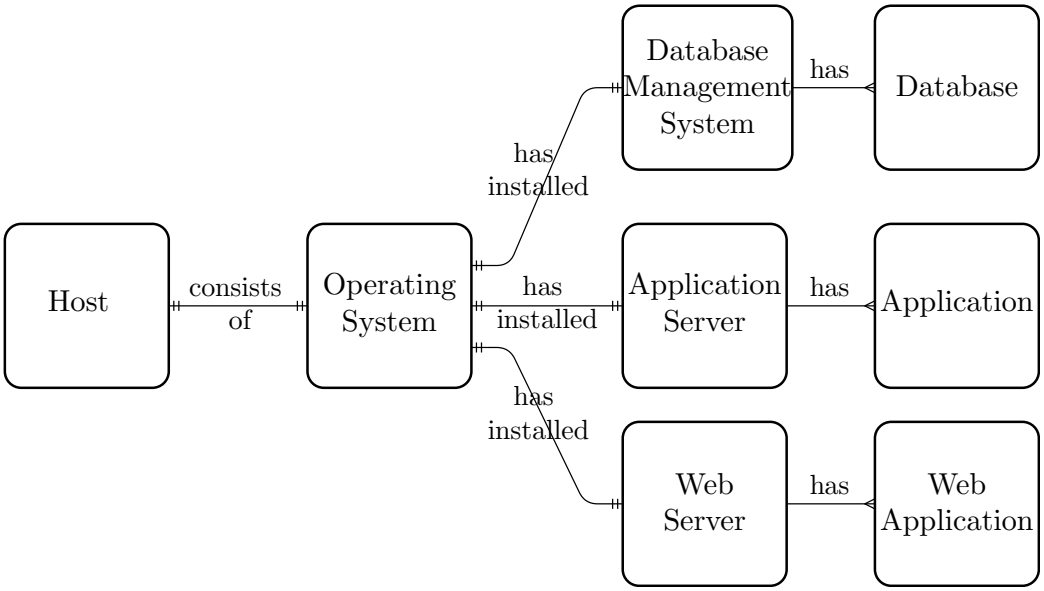


Figure 3.3: Example of configuration item tree.

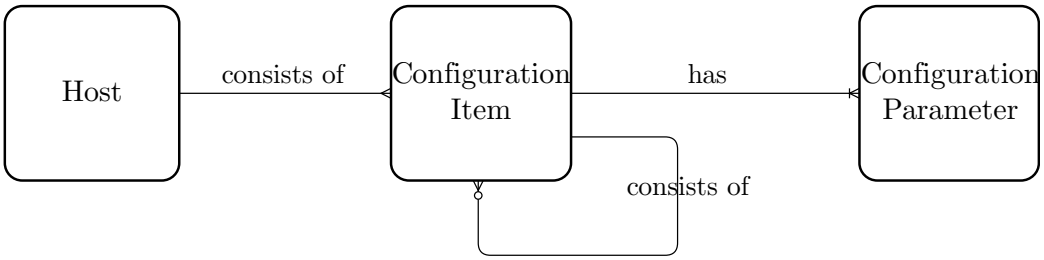


Figure 3.4: Host consists of several configuration items, which can further consist of smaller configuration items. The most granular configuration items have configuration parameters.

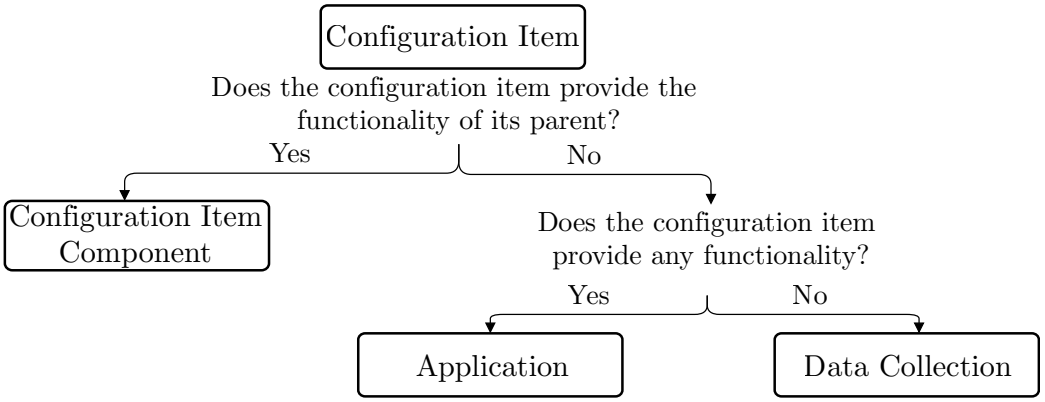


Figure 3.5: Classification of configuration items.

We classify configuration items into the following groups: *CI components*, *applications*, and *data collections*, depending on its functionality role. The classification is represented with the classification tree shown in Figure 3.5. The input is any of the configuration items of the configuration item tree that needs to be classified. If the configuration item provides functionality of the parent configuration item from configuration item tree, it is called *CI component*. Otherwise, further classification is needed: if the configuration item provides functionality, it is called *application*, otherwise, it is called *data collection*.

Next, we define dependency, which is shown in Figure 3.6.

Definition 3.2.6. Dependency is a relationship that exists between a configuration item A and a configuration item B when a configuration item A requires a service performed by a configuration item B in order to execute its function. [29]

Definition 3.2.7. When a configuration item A depends on a configuration item B (denoted as: $A \xrightarrow{\text{depends on}} B$), we say that A is the dependent and B is the antecedent [29].

A property, that applies to the dependencies between more than two configuration items, is called *transitive property*.

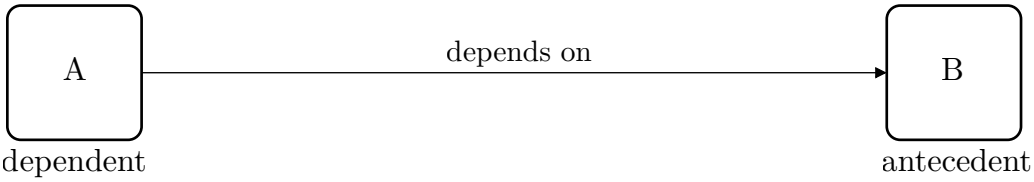


Figure 3.6: Dependency between configuration item A and configuration item B. We draw arrow from A to B, because A depends on B. A is called the dependent and B is called the antecedent.

Definition 3.2.8. If a configuration item A depends on a configuration item B, and a configuration item B depends on a configuration item C, then a configuration item A depends also on a configuration item C, as the the transitive property applies.

Transitive property also applies between configuration item trees of configuration items, which is defined in Definition 3.2.6.

Definition 3.2.9. If a configuration item A depends on a configuration item B, then all the children of a configuration item A with a configuration item A depends on all the children of a configuration item B and a configuration item B.

Definition 3.2.10. A graph, that contains only minimum number of dependencies, has a property called *transitive reduction*.

Other dependencies in a graph with transitive reduction property can be derived as defined in Definitions 3.2.8 and 3.2.9.

3.2.1 Dependency Classification

There are many dimensions on how to classify the dependencies as depicted in Figure 3.7. In this subsection we review them, give some examples of each one of them and discuss ones that are appropriate for our problem.

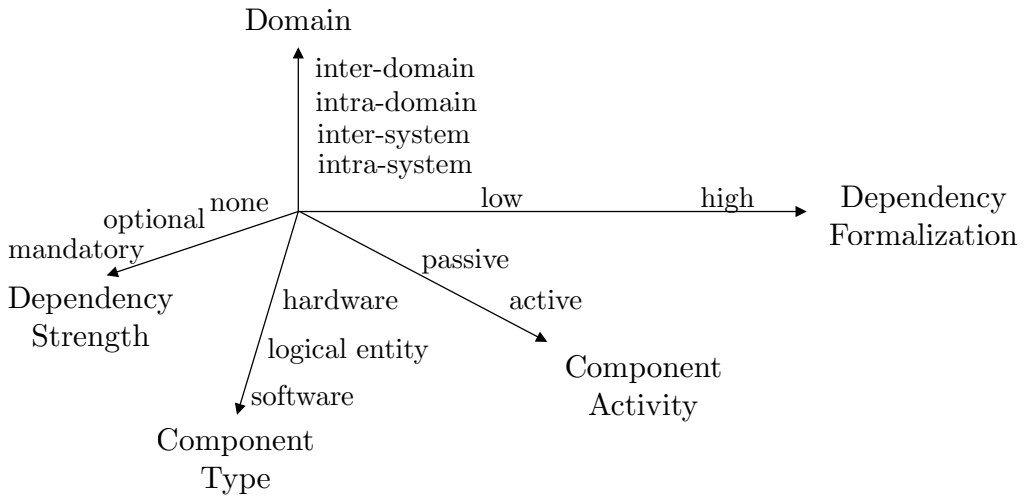


Figure 3.7: Multidimensional dependency classification [29]

Domain Dimension

Domain dimension tells us how "far" is the antecedent from the dependent (sharing memory space, sharing the same node, sharing the same subnet, being located within the same domain) [29].

Definition 3.2.11. Inter-domain dependencies are dependencies between different domains.

We treat each host as its own domain and therefore the dependencies between different hosts are called inter-domain. They also play a key role in affecting SLA management, since SLAs are usually associated with inter-domain dependencies.

Definition 3.2.12. Intra-domain dependencies are dependencies within same domain, meaning dependencies between different configuration items of a host.

All the installed applications depend on an operating system, operating system depends on a physical host and there can be some other dependencies between installed application (such as dependency between an application in application server and data collection).

Definition 3.2.13. Inter-system dependency is a dependency between a dependent service components on the same layer. [29]

An example of inter-system dependency is when a database client application depends on a database server.

Definition 3.2.14. Intra-system dependency is a dependency between an antecedent and a dependent component, if an antecedent component is located at a lower layer than dependent component. [29]

An example of intra-system dependency is a dependency between a web browser, which depends on the TCP service.

In this thesis we focus on discovering inter- and intra-domain dependencies. We do not consider inter- and intra-system because we are using a direct method, which does not monitor the traffic between different Open Systems Interconnection (OSI) layers nor it does rely on performance data as indirect methods do, as it is discussed in Section 2.2. Such dependencies are usually provided by APM tools as transaction breakdowns.

Component Activity Dimension

Component activity dimension tells us whether the antecedent is *active* (such as a piece of hardware or software) and can be directly/explicitly queried, or *passive* (such as a file), which by itself cannot be queried or instrumented and must always have an "intermediary" that acts on behalf of it. [29]

We have defined a dependency as a link between two configuration items (Definition 3.2.6) and we classified configuration items into IT components, applications, and data collections. IT components and applications are mostly *active* (there might be some passive configuration items, such as configuration files), whereas data collections are *passive*.

Component Type Dimension

This dimension tells us what the antecedent component actually is, whether is a piece of hardware, an end system, a software package or a service, etc.

This distinction is important because different types of components tend to behave and fail differently. [29]

In our work we do not consider the component type because it is only important that a dependency between two configuration items exists or not in our component dependency graph. However, the component type can be used in RCA's Change history module. Each change is essentially a configuration parameter, whose value has changed. Each configuration parameter has a type (whether it is a hardware, a code or a configuration file, etc.). This information can be included in a reasoning mechanism, to rank them appropriately according to their expected behavior.

Dependency Strength Dimension

Dependency strength dimension tells us the strength between the dependent component and the antecedent component. This is useful for intermittent dependencies; these are dependencies when a component requires resource only for certain periods of time (e.g. doing a backup requires to have another disk attached only at the times when backup is running). Mandatory dependencies have the highest dependency strength. [29]

The dependency strength dimension is not so important for our framework. We assume that all dependencies are mandatory, because we are using a direct method and it would take significant amount of time to discover strengths of dependencies. For indirect methods this information would be much simpler to obtain, however the problem that indirect methods have, is that some of the dependencies, which occur rarely, are not captured. Moreover, when performing RCA, you would need to predict which of the configuration items that affected configuration item were active at that time.

Dependency Formalization Dimension

Dependency formalization dimension tells us what degree of formalization this dependency has and, thus, to which degree it can be determined automatically. This serves as a metric that helps to evaluate how expensive

and/or difficult it is to acquire and identify this dependency, represent it, and, track this dependency during the lifetime of the component. For a "formalized" dependency, the cost of dealing with it is lower than that for a non-automated or not-well-formalized one. [29]

In our framework, we are automatically extracting dependencies therefore we only discover formalized dependencies.

3.3 Dependency Mapping Framework

The dependency mapping framework, depicted in Figure 3.8, comprises the following components: hosts with installed agents, a database server, and an analytics server. Analytics server consists of the following modules: Probabilistic Dependency Matching algorithm and root cause analysis module.

In the following subsections each of the components is described in more details.

3.3.1 Agent

An agent is a piece of software, that is installed on all hosts on which we want to discover dependencies. Hosts are connected in a network (either local area network, wide area network or any other type that allows communication between hosts). Agent periodically scans its host and reports changes compared to the previous scan.

The rules that describe the recursive structure of configuration item tree of a host, are stored in a document, called application definition document. Application definition document also describes how to parse each of the predefined configuration parameters (for example, from configuration file, from the properties, from different monitoring tools, etc.). This document is manually created and supports predefined technologies only due to the specific and unique structure each of them has.

Agent follows these rules in order to discover the recursive structure of configuration items on the host and their corresponding configuration param-

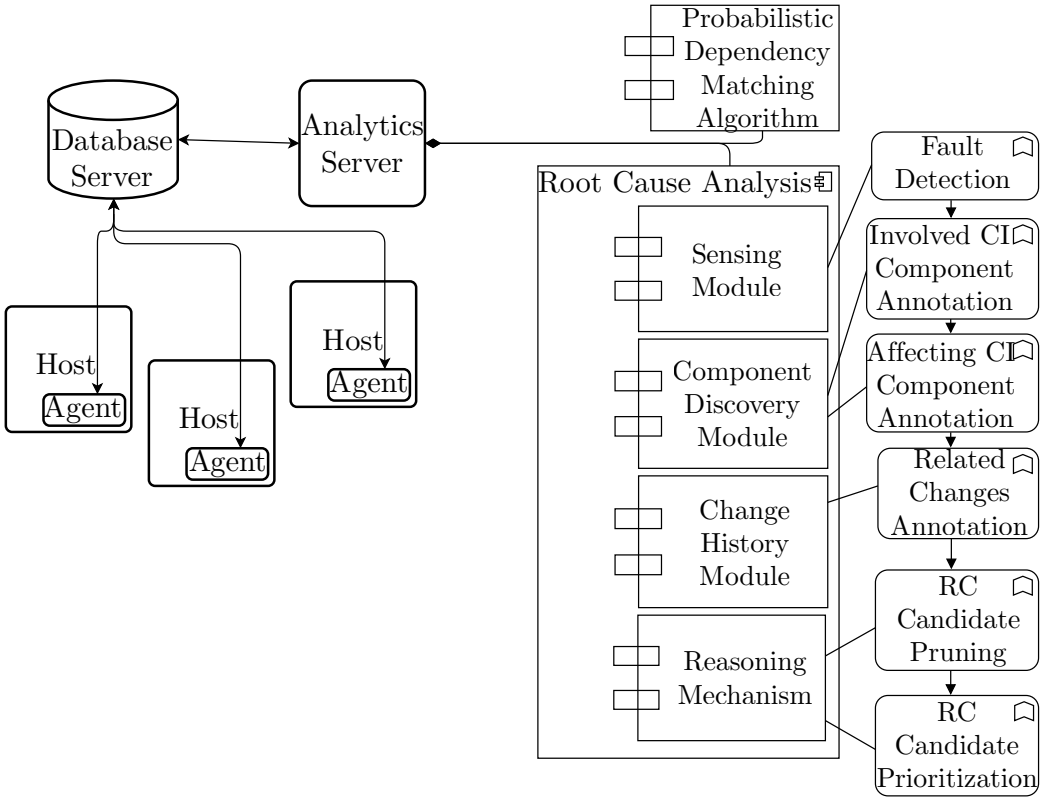


Figure 3.8: Dependency Mapping Framework

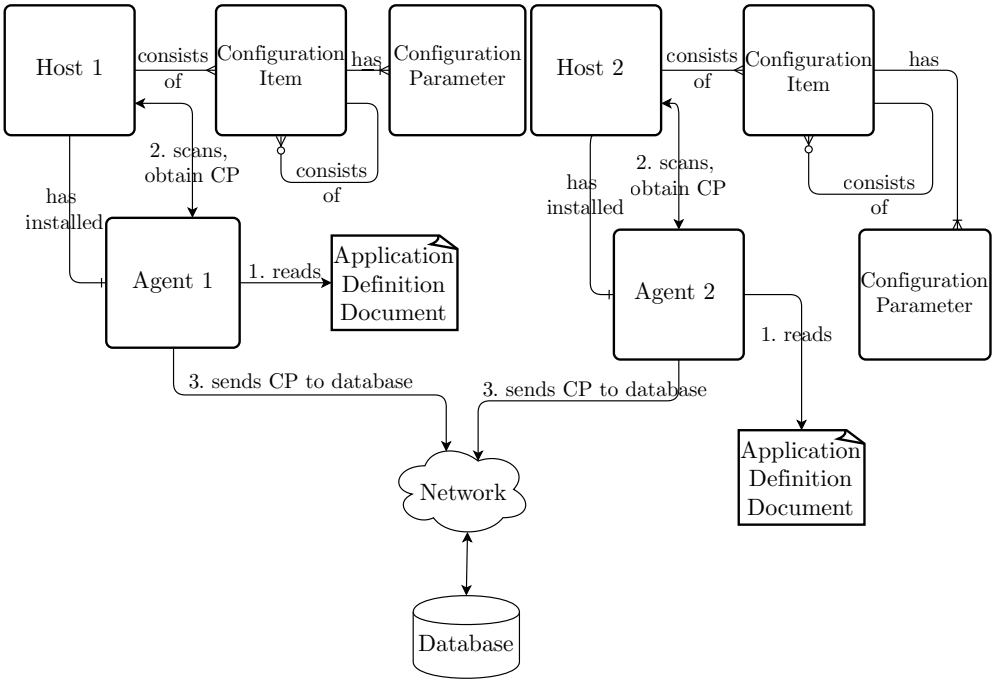


Figure 3.9: Agent framework comprises agents, installed on hosts. Each agent scans its host configuration item tree structure, described in application definition document and sends the configuration parameters of discovered configuration items through network to the database.

eters. When configuration parameters are discovered, they are stored in the database. Configuration parameters can also be collected upon occurrence of a change to the configuration parameters. Agent framework is depicted in Figure 3.9 and is summarized by [20].

Once the configuration parameters are obtained and stored in a database from a group of hosts on which we want to discover dependencies, we can start with the analysis of the data and the discovery of the dependencies among IT components on analytics server.

3.3.2 Analytics Server

Analytics server consists of the following modules: root cause analysis module and Probabilistic Dependency Matching algorithm.

The root cause analysis module consists of sensing module, component discovery module, change history module, and reasoning mechanism.

Sensing module detects any fault that happens in IT environment – either SLA violations or any other issues with IT components.

The component discovery module first correlates the fault with configuration item, denoted as *involved configuration item*. Secondly, it finds all the configuration items that involved configuration item depends on. For example, the fault on a web server could be caused by a database problem. Therefore, the module finds the database on which the web server depends on and includes both configuration items to the analysis. To extract the dependencies between different configuration items, the Probabilistic Dependency Matching algorithm plays the key role. It discovers all the dependencies between different configuration items and builds the component discovery graph.

The third module, change history module, finds all related changes on involved and affected configuration items and configuration parameters.

This list of configuration items is first pruned in the reasoning mechanism to eliminate non-relevant changes, and, later, prioritized by the likelihood of a change causing the fault. This list (or the top n suggestions) of the root cause candidate's changes is returned to the operator, which checks them and selects the appropriate actions to remedy the fault.

In the following chapter we present and describe the Probabilistic Dependency Matching, which identifies dependencies between components, encodes them into a component dependency graph, and allows the RCA module to effectively query dependencies during the RCA process.

Chapter 4

Probabilistic Dependency Matching

This chapter introduces the Probabilistic Dependency Matching (PDM) comprising several algorithms that are able to infer the dependencies between different architectural levels – between hosts, configuration items, and configuration parameters.

First, we present the input and the output of the algorithm. Secondly, we introduce the PDM's algorithms, which are able to extract dependencies on different levels. The algorithms also infer the probabilities of the dependencies. This chapter concludes with the analysis of the time complexity of the PDM's algorithms.

4.1 Input and Output

The input and the output of the PDM are directed graphs, more precisely directed cyclic graphs (DCG).

Definition 4.1.1. Directed graph G is a set of vertices V and a collection of directed edges E, denoted as $G = (V, E)$. Each directed edge connects an ordered pair of vertices. [36]

Definition 4.1.2. Directed cyclic graph (*DCG*) is a directed graph $G = (V, E)$ that contains at least one directed cycle. Directed cycle in a directed graph $G = (V, E)$ is a directed path with at least one edge whose first and last vertices are the same. A directed path in a directed graph G is a sequence of vertices in which there is a directed edge pointing from each vertex in the sequence to its successor in the sequence. [36]

In our thesis, vertices V corresponds to CIs and edges E corresponds to the dependencies between CIs. If a CI_A depends on a CI_B, we add a directed edge, pointing from CI_A to CI_B, as in Definition 3.2.6. There is one special case, where the edge E does not express the dependency, but an *instance*. These are edges between a generic CI (these are CIs with similar functionality, such as operating system, web server, application server, etc.) and their instances. These edges are denoted with *instance of*.

Each host's structure is expressed with its own DCG and is constructed in two steps. In the first step, we create a template DCG, which expresses generic dependencies between generic CIs. An example is depicted in Figure 4.1.

In the second step, we fill the generated template with host's data in the following way. For each configuration item, we find to which generic CI it corresponds to and append it to this node. For example, our host has Windows OS. We append this node to Operating Systems. Next, it has MySQL database management system with two schemas. We append MySQL node to Database Management System node, and both schemas to MySQL node. Similarly, we add IIS Web Server.

This example is shown in Figure 4.2, where the actual CIs the host consists of are shown with grey nodes. Notice that due to the hierarchical structure of the configuration items, which is described in application definition document, we already have some of the inter-domain dependencies.

The output of the algorithm are DCG with generated dependencies between configuration items. Each dependency has an additional property p – probability, which represents the likelihood that this dependency exists.

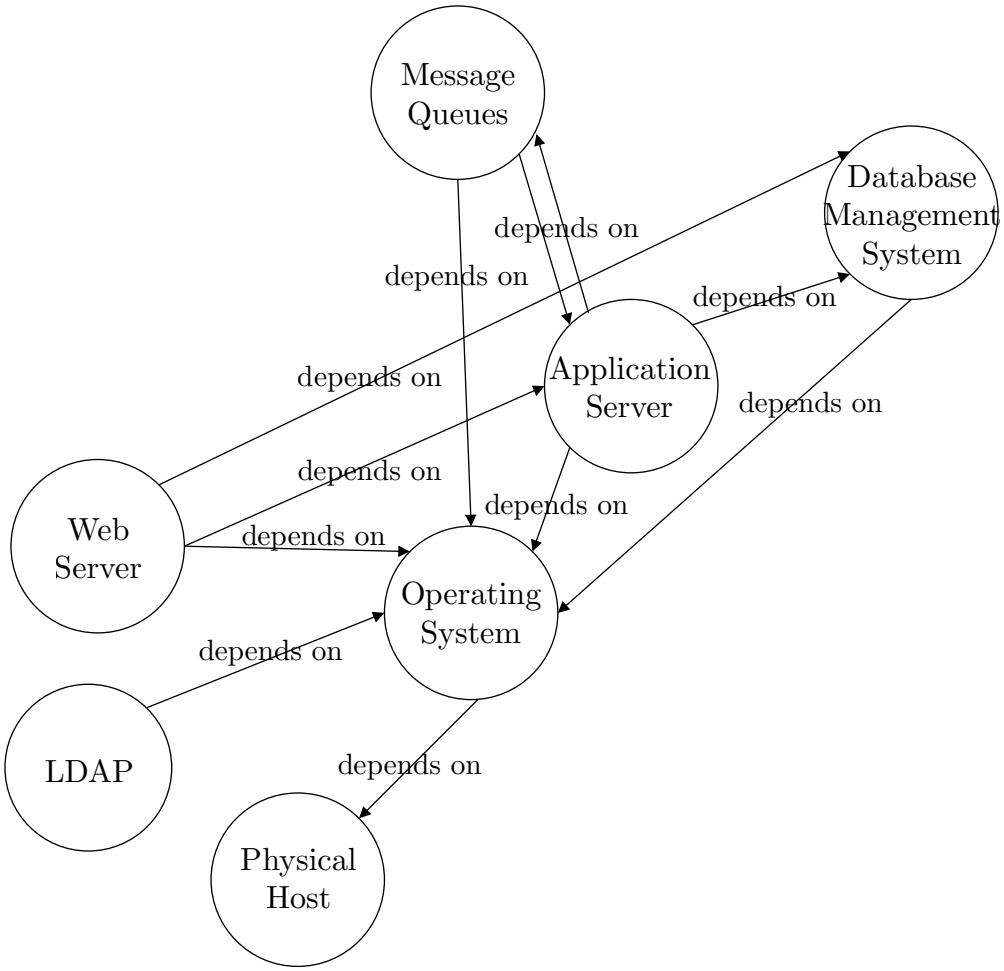


Figure 4.1: An example of template DCG, which depicts generic dependencies between operating systems, application servers, web servers, LDAPs, message queues, and database management systems.

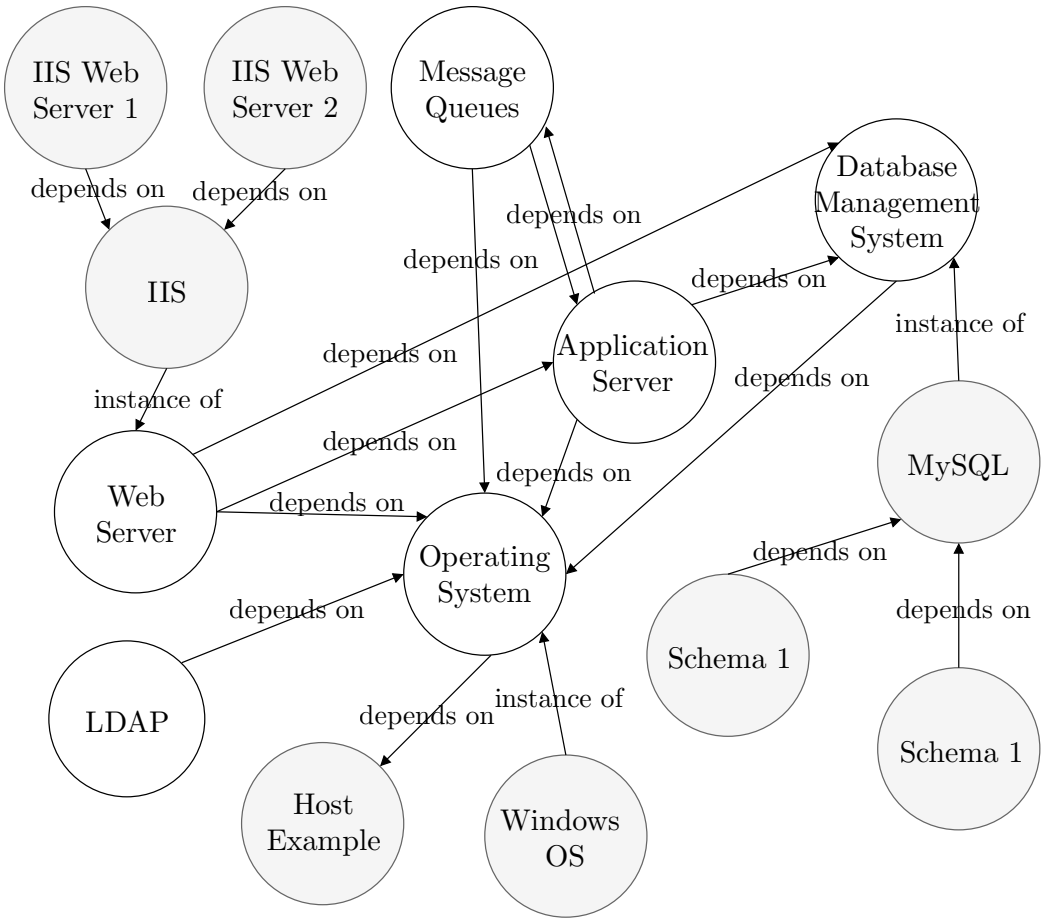


Figure 4.2: An example of host DCG, which consists of Windows OS, IIS web server, and MySQL database management system.

4.2 The Algorithms

There are different approaches that infer dependencies between configuration items. Naively, we can compare configuration parameters to each other, find the most likely configuration parameters responsible for dependency, and create a dependency between configuration items these configuration parameters correspond to. This approach makes $n \cdot (n-1)$ comparisons between configuration items, where n is the number of CIs; and almost $cp \cdot (cp-1)$ comparison between configuration parameters, where cp is the number of configuration parameters. We do not need to compare configuration parameters of the same configuration item.

For example, using the DCG depicted in Figure 4.2, we make $7 \cdot 6 = 42$ comparisons between IIS, IIS Web Server 1, IIS Web Server 2, MySQL, Schema1, Schema 2, and Windows OS nodes without any additional knowledge that DCG provides. The number of comparisons between configuration parameters is even larger due to the large number of configuration parameters comprising the configuration items.

In order to reduce the number of comparisons, we propose a PDM algorithm, which comprises the following four modules. The first module, called Host Dependency Extractor (HDE), extracts dependencies on the host level – it finds whether a host A depends on a host B. The second module, called Instance Dependency Generator (IDG), generates all possible dependencies between instances of generic CIs for all dependencies discovered by the first module. The third module, Configuration Item Dependency Extractor (CIDE), finds dependencies between CIs for each dependency generated by the second module. In addition, it assigns a probability that this dependency exists. The last module, called Instance Dependency Extractor (IDE), assigns the probabilities to dependencies between instances of generic CIs using the probabilities generated by the Configuration Item Dependency Extractor Module.

The proposed approach is also illustrated in the Figure 4.3.

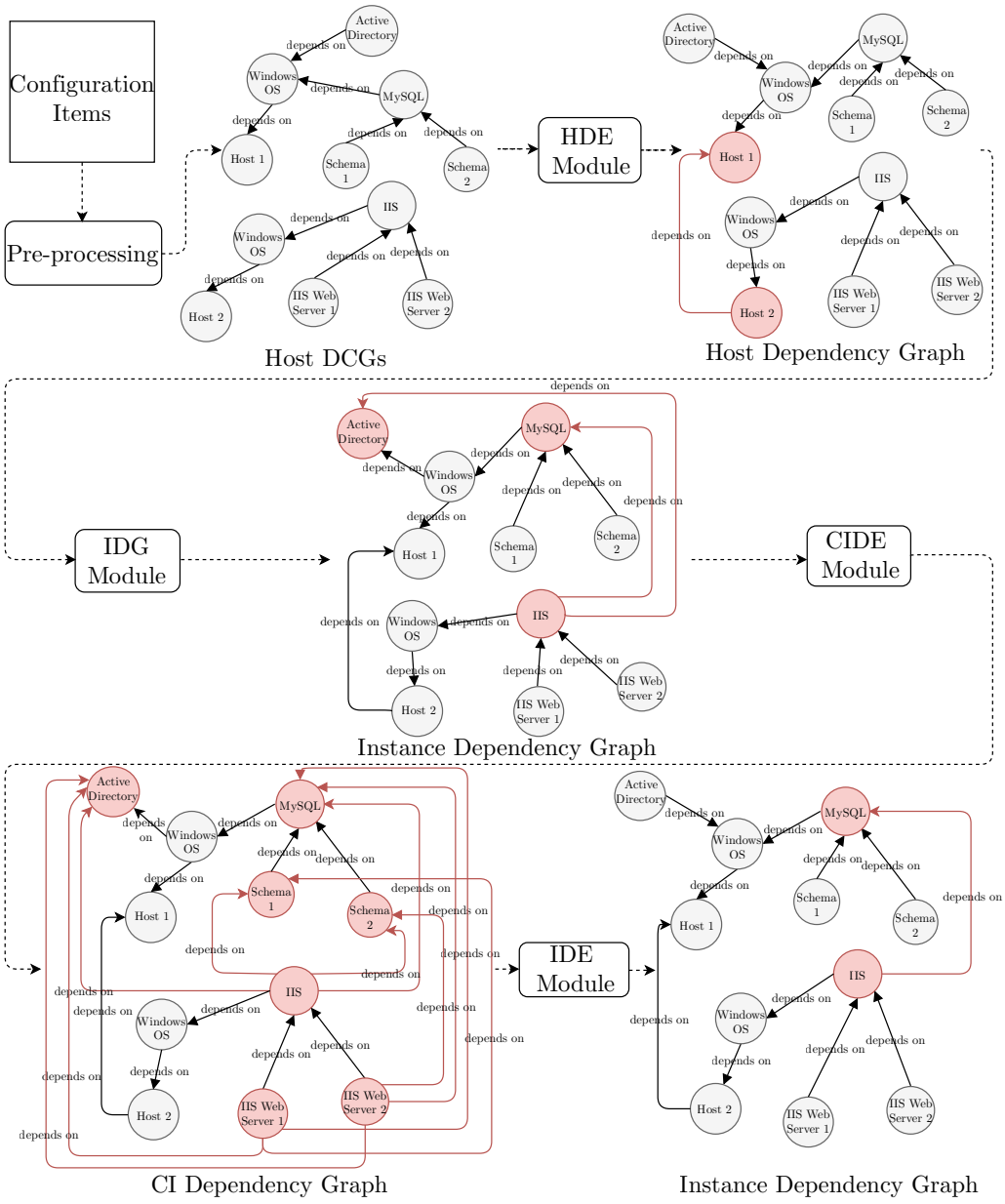


Figure 4.3: The proposed approach, which constructs the DCG from CIs, applies HDE, IDG, CIDE, and IDE Module in order to infer the dependencies between CIs.

4.2.1 Host Dependency Extractor Module

In this section we present Host Dependency Extractor (HDE) Module, which extracts dependencies at the host level. The underlying assumptions is that if a host A depends on a host B, host A must have a host B's property in its configuration parameters. However, reversed can be true as well. For example, if a host A grants FTP or SSH access to host B, host A's configuration parameters include host B's IP address. This means that host B depends on a host A because if host A goes offline, the FTP or SSH connection cannot be established. Such cases have to be handled separately with predefined rules.

As mentioned, one of the properties that distinguish a host from other hosts in the same network (or organization) is its IP address, we denote it as *source IP*. Another one is a host name, which is usually unique inside an organization. We denote the host name as *alias*. An agent's IP address corresponds to host's source IP and an alias of a host can be extracted via application definition document.

In order to extract dependencies between host A and host B, we do the following two steps. First, we traverse through configuration items of host A and check its configuration parameters. Secondly, if any of the configuration parameters includes either host B's source IP or its alias, we create a dependency between host A's root node and host B's root node with probability $p = 1$; unless the configuration parameter corresponds to one in the predefined rules. In such case we create a reversed dependency – a dependency between host B's root node and host A's root node with probability $p = 1$. We do the same for host B in order to extract dependencies pointing from host B to host A.

To apply this procedure to multiple hosts with installed agents, we initialize a list of objects *hosts*, which holds the hosts data (hosts's aliases, source IPs, and IDs), and object *rules*, which contains the configuration parameters for the reversed dependencies in advance. Also, we first iterate through all configuration parameters of all hosts and check each configuration parameter

for a dependency using *hosts* and *rules* objects. The pseudo-code for this procedure is presented with Algorithm 2.

Algorithm 2 Host Dependency Extractor Module

1: **for** $cp \in CP$ **do** ▷ Iterate through all configuration parameters
2: **for** $host_A \in hosts$ **do**
3: **if** ($host_A.alias \subset cp$ **or** $host_A.sourceIP \subset cp$) **then**
4: $host_B \leftarrow cp.host$
5: **if** $host_A \neq host_B$ **then** ▷ Create a dependency
6: **if** $cp \in rules$ **then**
7: $host_A \xrightarrow[p=1]{\text{depends on}} host_B$
8: **else**
9: $host_B \xrightarrow[p=1]{\text{depends on}} host_A$
10: **end if**
11: **end if**
12: **end if**
13: **end for**
14: **end for**

Once dependencies between hosts are extracted, the next step is to find the dependencies between CIs.

4.2.2 Instance Dependency Generator Module

In this module, called Instance Dependency Generator (IDG) Module, we generate all the possible candidate dependencies between instances of generic CIs in the following way. For a dependency between host A and host B (host A depends on host B), which is not a reversed dependency, we find which instance of generic CIs on host A includes host B's *source IP* or *alias* with function $GetInstanceCI(host_A, host_B)$. Next, we create a dependency from this instance to every other instance of generic CI of host B in order to create possible candidates of dependencies.

For a reversed dependency between host A and host B, we find an instance of general CIs on host B, which has host A's *source IP* or *alias* in configuration parameters with function $GetInstanceCI(host_B, host_A)$. After, we create a dependency between the host A's operating system, which is found with function $GetInstanceOS(host)$, and the host B's found instance.

Note that for function $GetInstanceCI(host_A, host_B)$ we do not need to search through all the configuration items. We can save the *source IP* or *alias* in the HDE Module to the nodes that it corresponds to. Therefore, we only need to iterate through the instances to find the right one.

Lastly, we generate intra-domain host dependencies in the following way. For a host A we create all possible pairs of dependencies between host A's instances of generic CIs.

Note that we could generate these candidate dependencies already in the HDE Module, but we decided to present them separately for easier understanding. The pseudo-code is presented in Algorithm 3. We use the same objects, *hosts* and *rules* as in HDE algorithm. The function $GetInstanceCIs(host)$ returns all the instances of the generic CIs that *host* consists of.

An example of the dependencies between two hosts created with this module is depicted in Figure 4.4.

4.2.3 Configuration Item Dependency Extractor Module

The Configuration Item Dependency Extractor (CIDE) Module extracts dependencies between configuration items and assigns each dependency a probability p that it exists. As we have seen in the IDG Module, it is easy to determine which CI includes other host's source IP or alias and harder to determine to which CI it actually belongs to. In order to find the right CI and to assign the probabilities, we search through the configuration parameters.

The underlying assumptions that hold for a dependency between CI_A and CI_B are the following. First, both CI_A and CI_B contain either a CI_B's

Algorithm 3 Instance Dependency Generator Module

for $host \in hosts$ **do**
  $instanceCIs \leftarrow GetInstanceCIs(host)$
  for $instanceCI_1 \in instanceCIs$ **do**
    for $instanceCI_2 \in instanceCIs$ **do**
      if $instanceCI_1 \neq instanceCI_2$ **then** ▷ Create candidate dependency
        $instanceCI_1 \xrightarrow{\text{depends on}} instanceCI_2$
      end if
    end for
  end for
end for
for $host_{dep} \in hostDependencies$ **do**
  $host_A \leftarrow host_{dep}.dependent$
  $host_B \leftarrow host_{dep}.antecedent$
  if $host_{dep}$ generated by $rules$ **then**
    $instanceCI_A \leftarrow GetInstanceCI(host_B, host_A)$
    $instanceCI_B \leftarrow GetInstanceOS(host_A)$
    $instanceCI_B \xrightarrow[p=1]{\text{depends on}} instanceCI_A$
  else
    $instanceCI_A \leftarrow GetInstanceCI(host_A, host_B)$
    $instanceCIs_B \leftarrow GetInstanceCIs(host_B)$
    for $instanceCI_B$ in $instanceCIs_B$ **do**
      $instanceCI_A \xrightarrow{\text{depends on}} instanceCI_B$
    end for
  end if
end for

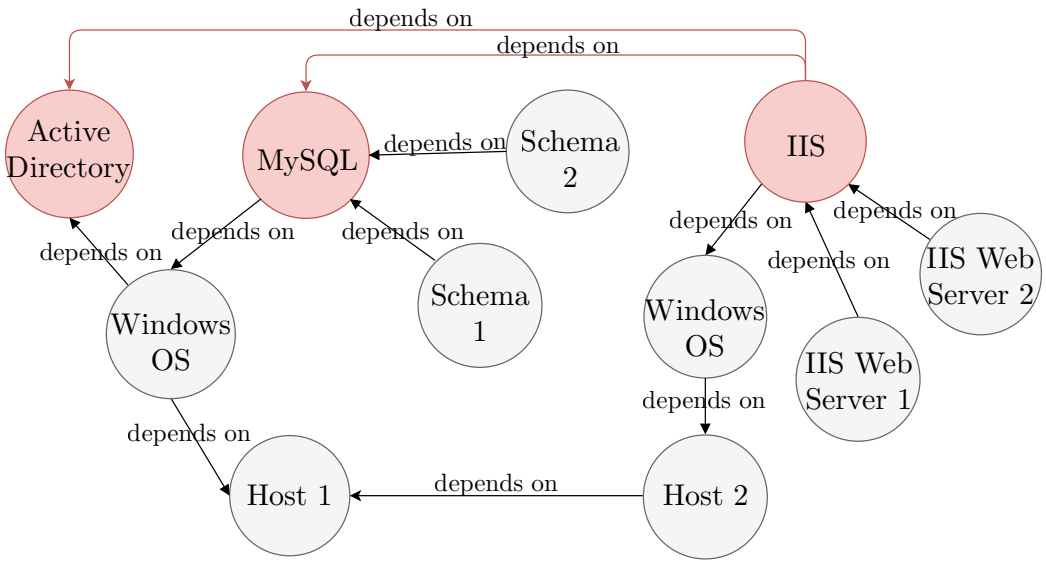


Figure 4.4: An example of candidate dependencies created with IDG module.

host alias or source IP. Second, if a CI_A depends on CI_B, CI_A has some information about CI_B in its configuration parameters, such as a CI_B name. CI_B name can be either a database name, an application server's application name, etc. However, not all CI names are good candidates. Often, we can find names that are common – such as default databases names, default applications names; which are not deleted after installation. We would like to disregard them since they are not very indicative.

We use weighting method based on *inverse document frequency (idf)*, proposed by Karen Sparck Jones [27], which comes from the information retrieval field. In order to calculate idf score of some specific term in a collection of documents, we first need to calculate the *document frequency* of the term t. Document frequency $df(t)$ is defined as a number of documents that contain term t.

The idf score of a term t in a collection of documents is calculated as

$$idf(t) = log(\frac{N}{df(t)}),$$

where N is the number of documents in a collection. For very rare terms,

idf terminology	our domain terminology
collection	technology
document	instance with sub-configuration tree
term	CI name

Table 4.1: Mapping between idf terminology and terminology used in our domain.

the idf score is very high and is low for common, frequent words.

In order to use idf score in our domain, we define what term, document, and collection corresponds to our domain. A collection corresponds to a specific technology, such as Windows operating system, IIS web server, Oracle Database. Therefore the number of collections is the same as the number of different technologies. Each collection is composed of documents. A document is an instance of a collection's technology on a host along with the instance's sub-configuration item tree. A collection for a specific technology therefore comprises of all instances, that corresponds to this technology. A term corresponds to a (delimited) CI name. Therefore we calculate an idf score for each term for each technology separately. The mapping between idf and our domain terminology is presented in Table 4.1.

The algorithm for CIDE Module is presented in Algorithm 4. For each *dependency* from *dependencies*, that were generated by the Algorithm 3, it generates all the transitive dependencies with the function *GetAllTransitiveDependencies*(*dependency*), as defined in Definition 3.2.9. Next, it takes a transitive dependency and gets antecedent and dependent CIs. Furthermore, it checks whether one of the antecedent terms is included in dependent's configuration parameters and calculates an idf score. It also checks whether both CIs include antecedent's host alias or source IP. Using idf score, inclusion of antecedent's host information in both CIs and some additional features, it calculates the probability of a dependency with function *CalculateProbability*(*dependency*). The probability calculation is presented in the next section, which also lists all the features that are used.

Algorithm 4 Configuration Item Dependency Extractor Module

Require: $GetTerms(antecedent)$ ▷ It returns the terms created from delimited *antecedent*'s name

Require: $GetCollection(technology)$ ▷ It return the collection for specific technology

Require: $GetIdfScore(term, collection)$ ▷ It returns the idf score of a $term \in collection$

1: **for** $dep \in dependencies$ **do**
2: **if** *dep* not generated by *rules* **then**
3: $transitiveDependencies \leftarrow GetAllTransitiveDependencies(dep)$
4: **for** $transitiveDependency \in transitiveDependencies$ **do**
5: $dependent \leftarrow transitiveDependency.dependent$
6: $antecedent \leftarrow transitiveDependency.antecedent$
7: $terms \leftarrow GetTerms(antecedent)$
8: $maxIdfScore \leftarrow -1$
9: **for** $term \in terms$ **do**
10: **if** $term \subset dependent.CP$ **then** ▷ Check if term is in dependent's configuration parameters
11: $idfScore \leftarrow GetIdfScore(term, GetCollection(dependent.technology))$
12: **if** $idfScore > maxIdfScore$ **then**
13: $maxIdfScore \leftarrow idfScore$
14: **end if**
15: **end if**
16: **end for**
17: **if** $antecedent.hostName \subset dependent.CP$ **and** $antecedent.hostName \subset antecedent.CP$ **then**
18: $transitiveDependency.HostMatching \leftarrow True$
19: **end if**
20: $transitiveDependency.idfScore \leftarrow maxIdfScore$
21: $probability \leftarrow CalculateProbability(transitiveDependency)$
22: $dependent \xrightarrow[p=probability]{\text{depends on}} antecedent$
23: **end for**
24: **end if**
25: **end for**

4.2.4 Probability Calculation

In this subsection we present a supervised, machine learning algorithm that assigns a probability of a dependency, given a set of features, called Naive Bayes [18]. We denoted this function as $CalculateProbability(dependency)$ in the Algorithm 4.

Naive Bayes is a probability model that learns conditional probabilities of each of the n features $\mathbf{x} = (x_1, x_2, ..., x_n)$ for each of m classes or outcomes C_m:

$$P(C_m|x_1, x_2, ..., x_n).$$

We can reformulate above equation using Bayes' theorem [12]:

$$P(C_m|\mathbf{x}) = \frac{P(C_m)P(\mathbf{x}|C_j)}{P(\mathbf{x})}. \tag{4.1}$$

If we focus on numerator term in Equation 4.1, we can use the chain rule to derive:

$$\begin{aligned} P(C_m)P(\mathbf{x}|C_m) &= P(x_1, ..., x_n, C_m) \\ &= P(x_1|x_2, ...x_n, C_m)P(x_2|x_2, ..., x_n, C_m)...P(x_n|C_m)P(C_m) \end{aligned} \tag{4.2}$$

The problem with the Equation 4.2 is that these conditional probabilities are usually hard to estimate. Fortunately, the computation becomes feasible with Naive Bayes independence assumption: each feature x_i is conditionally independent of every other feature x_j, where $x_i \neq x_j$ given the class C_m. Therefore, we can rewrite the Formula 4.2 as:

$$\begin{aligned} P(C_m)P(\mathbf{x}|C_m) &= P(x_1|C_m)P(x_2|C_m), ..., P(x_n|C_m)P(C_m) \\ &= P(C_m)\prod_{i=1}^{n} P(x_i|C_m) \end{aligned} \tag{4.3}$$

We only need to derive the denominator term of Equation 4.1 using normalization as described by Russell and Norvig [35]. Normalization uses the

conditional independence assumption and conditional probabilities to derive the following formula:

$$P(\mathbf{x}) = \sum_{j=1}^{m} \Big(P(C_j) \prod_{i=1}^{n} P(x_i|C_m) \Big) \tag{4.4}$$

The Equation 4.1 can be thus written:

$$P(C_m|\mathbf{x}) = \frac{P(C_m) \prod_{i=1}^{n} P(x_i|C_m)}{\sum_{j=1}^{m} \Big(P(C_m) \prod_{i=1}^{n} P(x_i|C_m) \Big)}. \tag{4.5}$$

In order to use Naive Bayes in our domain, we need to define what classes and features represent. We have binary classification problem – we predict whether a dependency exists or not. The features that we consider are the following:

- *Dependency domain*: whether a dependency is inter- or intra-domain
- *Template*: whether a dependency follows the rules we describe in the template
- *Number of documents*: number of documents in collection
- *idf score*: idf score of a term, used in dependent's technology
- *Host matching*: it tells us whether an antecedent's host *alias* or *source IP* appear in dependent and antecedent configuration parameters.
- *Host connection*: it tells us whether both hosts of configuration items are connected with a dependency

In order to estimate the probability of a dependency, we have to estimate the conditional probabilities $P(x_i|C_m)$ and probabilities of a class $P(C_j)$, which are also called prior probabilities, on a smaller sample of our dataset – called training set.

However, before we can estimate, we have to preprocess the features.

Feature Preprocessing

Dependency domain, *Template*, *Host matching*, and *Host connection* are binary features, *Number of documents* and *idf score* are continuous. To use the latter ones as binary features, we need to do the data binning, where we will get from each feature k features, where k presents number of bins. By example, if the data ranges from 0 to 1, we could just create 9 bins or features: first contains values from 0 to 0.1, second contains values from 0.1 to 0.2 and so on. However this is not the best approach if the majority of values is centered in the last bin.

There are many other approaches to discretize continuous data, such as uniform binning, entropy based, and purity based approach. The article by Dougherty et al. [17] compares different approaches and shows that entropy based approach performed better and it even improved the performance of Naive Bayes algorithm, therefore we use entropy based approach to bin our continuous features.

Entropy based discretization [30] is based on entropy, or expected information. Let S be an object of a feature and class labels, consisting of m classes with probabilities $p_1, ..., p_m$. The entropy of such object is defined as:

$$E(S) = \sum_{i=1}^{m} p_i \log_2(\frac{1}{p_i}).$$

Let A be a feature (a value) that divides an object S into disjoint subsets $S1, S2, ..., S_n$. The entropy of partitioned S by a feature A is defined as:

$$E(A, S) = \sum_{i=1}^{n} \frac{|S_i|}{|S|} E(S_i),$$

where $|X|$ denotes the cardinality of a set X.

Information gain is used for evaluating the importance of a feature A on classification and is calculated as:

$$InformationGain = E(S) - E(A, S).$$

In practice we bin the data into subsets in the following way. For each potential split, we calculate the information gain of it. We select the split with highest information gain and recursively partition until some termination criteria is reached. This can be either the number of bins or when the bin's entropy value is below some threshold.

Once the features are discretized, we can calculate the conditional probabilities $P(x_i|C_m)$ in order to compute the probability of dependency $P(C_m|\mathbf{x})$.

4.2.5 Instance Dependency Extractor Module

The last module of PDM, called Instance Dependency Extractor (IDE) Module, assigns the probability to each dependency generated by the IDG Module. It uses the information calculated in the CIDE Module in the following way. The CIDE Module generates all the transitive dependencies between antecedent's and dependent's configuration item tree with the function *GetAllTransitiveDependencies*(*dependency*) and assigns them the probabilities. IDE finds the maximum probability among the transitive dependencies, and assigns it to a dependency, generated by the IDG Module.

The pseudo-code is presented in the Algorithm 5.

4.3 Time Complexity Analysis

In this section we provide an analysis of the time complexity of each module our algorithm consists of. We use the following variables throughout the analysis:

- h – the number of hosts
- cp – the number of all configuration parameters
- ci – the number of all configuration items
- $inst$ – the number of instances

Algorithm 5 Instance Dependency Extractor Module

```
1: for dependency ∈ IDGdependencies do
2:     if dependency not generated by rules then
3:         transitiveDependencies ← GetAllTransitiveDependencies(dependency)
4:         maxProbability ← 0
5:         for transitiveDependency ∈ transitiveDependencies do
6:             if transitiveDependency.probability > maxProbability then
7:                 maxProbability ← transitiveDependency.probability
8:             end if
9:         end for
10:        dependency.probability ← maxProbability
11:    end if
12: end for
```

- $host_{dep}$ – number of dependencies between hosts
- $children(CI)$ – number of children of CI in configuration item tree
- d_i – number of dependencies between instances
- cp_{ci_i} – number of configuration parameters of configuration item ci_i
- $cp_{h_i} = \frac{cp}{h}$ – average number of configuration parameters on the host h_i
- $ci_{h_i} = \frac{ci}{h}$ – average number of configuration items on the host h_i
- $inst_{h_i} = \frac{inst}{h}$ – average number of general instances on the host h_i

4.3.1 Analysis of HDE Module

The algorithm HDE runs through all the configuration parameters, checks if any of the configuration parameters contains either an alias or source IP, and creates a dependency between a host of a configuration parameter and found host if they are not the same. This algorithm hence runs with time

complexity:

$$\mathcal{O}(cp \cdot h \cdot h), \tag{4.6}$$

where $h \ll cp$. The first h in Equation 4.6 refers to the line 2 in Algorithm 2, where we iterate through all of the host's aliases and source IPs. The second is for creating a dependency – we need to find $host_A$ and $host_B$ in the database and create a dependency between them. If we use an indexed database, we can find a host in $log(h)$, therefore the time complexity is $\mathcal{O}(cp \cdot h \cdot log(h)) \approx \mathcal{O}(cp)$.

4.3.2 Analysis of IDG Module

The IDG Module creates intra- and inter-domain dependencies separately. For generating dependencies between instances for intra-domain dependencies, we iterate through all the hosts and create all possible dependency pairs between the instances. Therefore, it runs with time complexity: $\mathcal{O}(h \cdot inst_{h_i}^2)$. We assume that accessing instances can be done in $\mathcal{O}(1)$, if instances are saved on a *host* object.

When we generate inter-domain dependencies, we first iterate through all dependencies, generated by HDE, find the instance i that contains *source IP* or *alias* (in order to not to iterate through all configuration parameters again, we can save it in the HDE already), and generate dependencies between instance i and all instances on other host. Therefore, it runs in $\mathcal{O}(host_{dep} \cdot (inst_{h_i} + inst_{h_i})) = \mathcal{O}(host_{dep} \cdot inst_{h_i})$.

The time complexity of Instance Dependency generator is:

$$\mathcal{O}(h \cdot inst_{h_i}^2 + host_{dep} \cdot inst_{h_i}).$$

Because we efficiently reduced the search space for finding dependencies between hosts in the first step, the number of dependencies between hosts is therefore $host_{dep} \ll h^2$.

4.3.3 Analysis of CIDE Module

CIDE Module traverses through all the candidate dependencies, generated by the Algorithm 3. For each candidate dependency, it gets all transitive dependencies, which results with $\mathcal{O}(\frac{ci_{h_i}}{inst_{h_i}} \cdot \frac{ci_{h_i}}{inst_{h_i}})$. Then, it splits the antecedent's name to terms and searches for each term if it is contained in CP of dependent, and calculates idf score. It also checks if antecedent's host name is included in dependent and antecedent's configuration items. After, it calculates a probability and assigns this value to dependency. The probability calculation can be pre-calculated to save time.

Therefore, the time complexity of this algorithm is:

$$\mathcal{O}((h{\cdot}inst_{h_i}^2{+}host_{dep}{\cdot}inst_{h_i}){\cdot}(\frac{ci_{h_i}}{inst_{h_i}})^2{\cdot}cp_{ci_i}{\cdot}C) = \mathcal{O}(C{\cdot}cp_{ci_i}{\cdot}ci_{h_i}^2{\cdot}(h{+}\frac{host_{dep}}{inst_{h_i}})).$$

Iteration through terms and searching through configuration parameters is presented with the constant C.

4.3.4 Analysis of IDE Module

IDE Module first traverses through all candidate dependencies of IDG Module, next, it traverses through all transitive dependencies generated by the CIDE Module. It finds the maximum probability between transitive dependencies and assigns this probability to the candidate dependency. The time complexity of this algorithm is:

$$\mathcal{O}((h \cdot inst_{h_i}^2 + host_{dep} \cdot inst_{h_i}) \cdot (\frac{ci_{h_i}}{inst_{h_i}})^2) = \mathcal{O}(h \cdot ci_{h_i}^2 + host_{dep} \cdot \frac{ci_{h_i}^2}{inst_{h_i}}).$$

All together, the time complexity of the PDM algorithm is:

$$\begin{aligned} \mathcal{O}(cp \cdot h \cdot log(h) &+ (h \cdot inst_{h_i}^2 + host_{dep} \cdot inst_{h_i}) \\ &+ C \cdot cp_{ci_i} \cdot ci_{h_i}^2 \cdot (h + \frac{host_{dep}}{inst_{h_i}}) \\ &+ h \cdot ci_{h_i}^2 + host_{dep} \cdot \frac{ci_{h_i}^2}{inst_{h_i}}). \end{aligned} \tag{4.7}$$

The largest terms in Equation 4.7 are the first one, $\mathcal{O}(cp \cdot h \cdot log(h))$ of HDE Module, and the third one $C \cdot cp_{ci_i} \cdot ci_{h_i}^2 \cdot (h + \frac{host_{dep}}{inst_{h_i}})$ of CIDE Module due to the following reasons. Firstly, the last term of IDE Module and the second term of IDG Module are already included in CIDE Module. Secondly, the number of all configuration parameters is significantly larger than the number of all configuration items. The number of configuration items is a bit larger than the number of instances, and the number of hosts is smaller than the number of instances. Therefore the terms that search through configuration items prevail, however it is difficult to estimate if the first or the last term is bigger. It depends on how many searches through configuration parameters we perform in the Algorithm 4. Therefore, the time complexity of PDM is:

$$\mathcal{O}(cp \cdot h \cdot log(h) + cp_{ci_i} \cdot ci_{h_i}^2 \cdot (h + \frac{host_{dep}}{inst_{h_i}})).$$

If one would naively compare each configuration parameter with each other, the time complexity would be quadratic in terms of the size of configuration parameters, $\mathcal{O}(cp^2)$. With our approach we do not reach $\mathcal{O}(cp^2)$ due to the significant reduction of the search space by the HDE Module.

Chapter 5

Experimental Results

In this chapter we first describe the experimental setup and the implementation details. Next, we present the methodology for evaluation, which also describes the third scientific contribution of the thesis. This chapter concludes by demonstrating the efficiency of the proposed algorithms and compares them to the naive approach.

5.1 Experimental Setup

The proposed algorithm was evaluated with a dataset, which was obtained from a large enterprise, consisting of 115 hosts. These hosts serve various business applications and apply to different environments, such as development, testing, pre-production, and production. All the hosts together contain more than 200 instances, and more than 5,000 configuration items. This presents with more than 500,000 configuration parameters. In total, there are nine different technologies supported by these hosts.

5.1.1 Implementation Details

We designed the architecture depicted in Figure 5.1 for storing the data and running the proposed algorithms. The architecture consists of two databases and PDM algorithm. The first database is a graph database, which stores

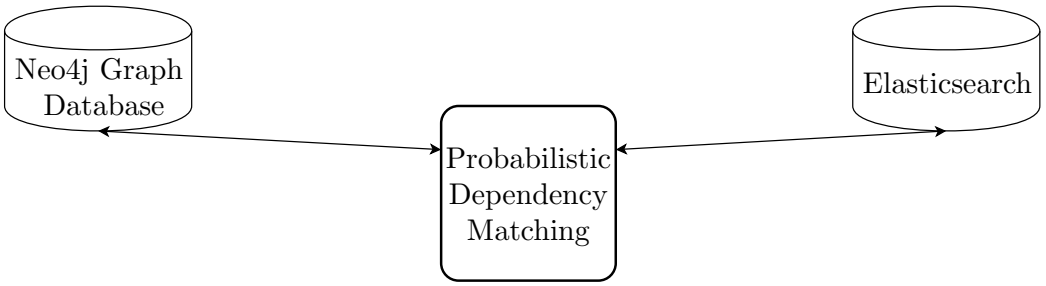


Figure 5.1: Implementation architecture.

configuration item trees of hosts as a DCG. For our purposes we used Neo4j graph database [9], which also provides basic visualization. An example of a host stored in a Neo4j graph database is shown in Figure 5.2.

The second database stores configuration parameters of the configuration items. For our implementation we used ElasticSearch database [8], which features faster retrieval and search through extensive amount of configuration parameters compared to a traditional SQL database.

5.2 Evaluation Methodology

The proposed algorithm consists of several modules, as described in Chapter 4, where each module produces dependencies on different architectural level. Regardless of the level, we evaluate the performance of modules the same way as follows. We compare the predicted dependencies to the real dependencies, which were labeled by a domain expert. However, there are different comparison techniques to evaluate if predicted and real dependencies match. We present and define three: Basic, Transitive, and Root cause evaluation technique, which adjusts the evaluation for the root cause analysis task. The latter two are also the third scientific contribution of this thesis.

Note that intra-domain dependencies resulting from the hierarchical structure of the application definition document are not included in the performance measures as they always reflect the real dependencies.

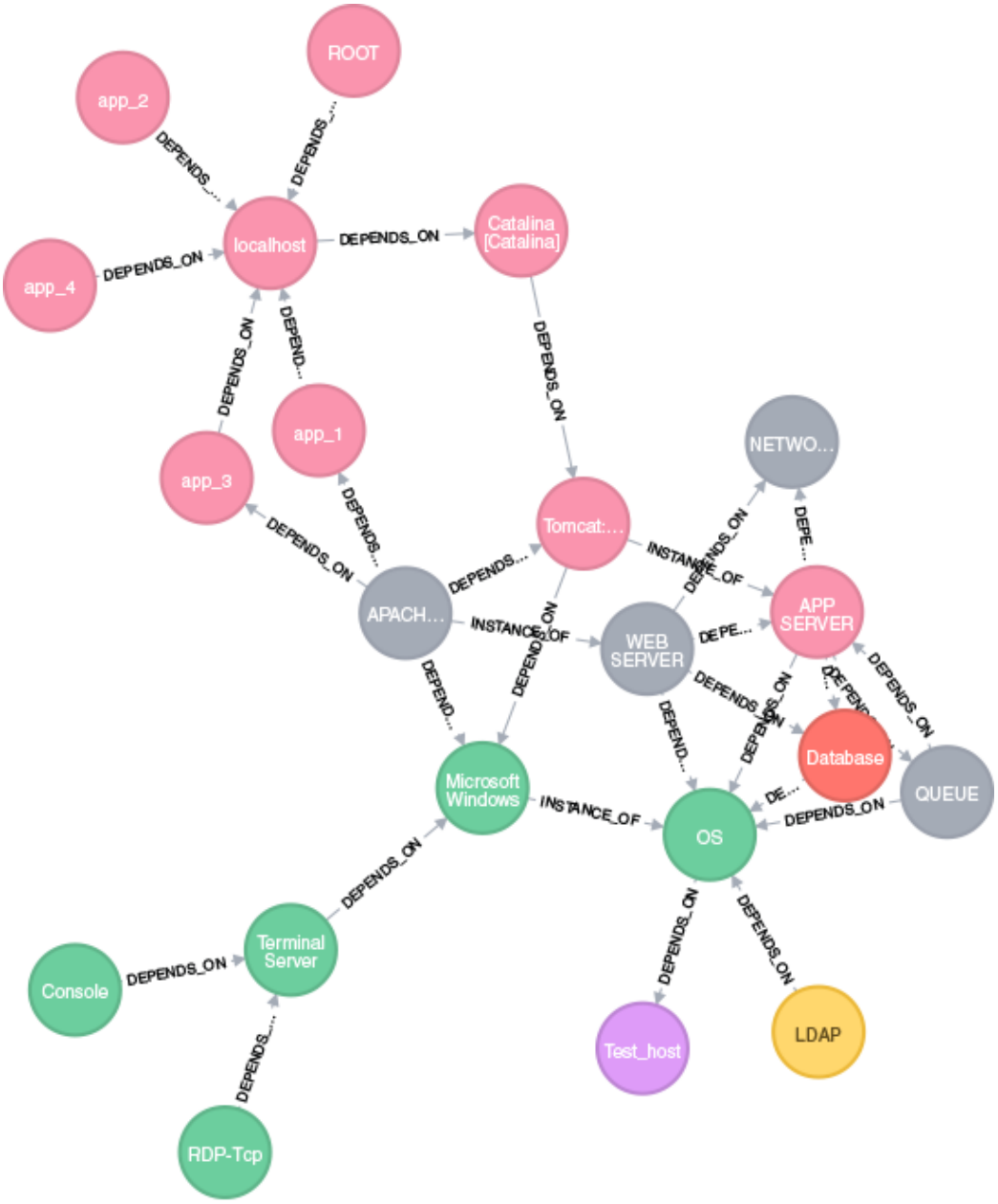


Figure 5.2: A sample screenshot of a host stored as DCG in Neo4j graph database.

5.2.1 Basic Evaluation Technique

Performance measures that are used in the evaluation include the following terms:

- **True positive (TP) dependency** is a dependency that is predicted as true and is labeled as true one.
- **False positive (FP) dependency** is a dependency that is predicted as true but is labeled as false one.
- **False negative (FN) dependency** is a dependency that is predicted as false but is labeled as true one.

The performance measures are calculated as follows.

Definition 5.2.1. Precision is the fraction of correctly identified dependencies among all dependencies that are predicted as true dependencies:

$$precision = \frac{TP}{TP + FP}$$

.

Definition 5.2.2. Recall is the fraction of correctly identified dependencies among all true dependencies:

$$recall = \frac{TP}{TP + FN}$$

.

The measure that computes a harmonic mean of the precision and recall, is called *F-measure*. It reaches the best value at 1 and worst at 0.

$$F\text{-}measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

.

Given the highly unbalanced dataset, which contains only a small amount of true dependencies against a high amount of false dependencies, performance measure accuracy is not suitable.

For Basic evaluation technique, we first count the number of TP, FP, and FN dependencies and calculates precision, recall, and F-measure.

The other two techniques count the TP, FN, and FN differently, as described in the following sections.

5.2.2 Transitive Evaluation Technique

Basic evaluation technique does not take into account the transitive properties, as defined in Definitions 3.2.8 and 3.2.9. The transitive role becomes more important for evaluation of dependencies between different configuration items. Also, predicted and labeled dependencies can come from a transitive reduction graph, and therefore the transitive property can be taken into account.

An example of the dependencies generated with transitive property is depicted in Figure 5.3, where the initial dependency is drawn with the red arrow and its transitive dependencies are drawn with green arrows. The importance of the transitive property can be explained with the following example. The real dependency is drawn with the red arrow – between IIS Web Server 2 and MySQL as shown in Figure 5.3. We predicted only a dependency between IIS Web Server 2 and Schema 2. Using the Basic evaluation technique, we count such dependency as FP, even though the predicted dependency is a part of the real dependency. Using Transitive evaluation technique, we have one TP and two FN, which results with 33% recall and 100% precision.

To evaluate the dependencies with this evaluation approach, we first generate all the transitive dependencies for predicted dependencies. Each generated dependency is also predicted dependency. We apply the same procedure to the real dependencies as well. One can note that the number of predicted and real dependencies generated by the transitive property is much larger than without the transitive property. Next, we count the number of the TP, FN, FP dependencies of the transitively generated dependencies and calculate precision, recall, and F-measure.

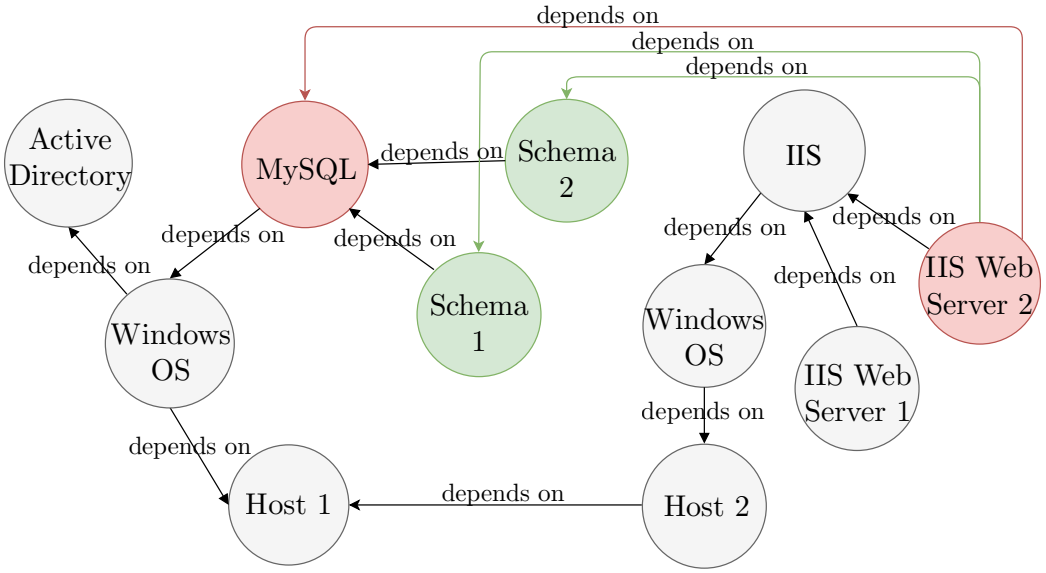


Figure 5.3: An example with initial dependency drawn with red arrow and transitive dependencies, drawn with green arrow.

5.2.3 Root Cause Evaluation Technique

This evaluation technique takes into an account the following properties of dependencies, that are essential for RCA. First, the dependency d_i is more important if there is a high number of transitive dependencies going through d_i. Secondly, if a dependency d_i is the only dependency that connects two vertices, and the path that connects both vertices is long, it should be more important.

We can express the importance of a dependency d_i with a weight $w(d_i)$ – if a dependency d_i is more important, it should get a higher weight $w(d_i)$. The weight should take into account both properties. Therefore, it is proportional to the number of shortest paths between vertices V_m and V_n that go through d_i, and the longest shortest path between vertices V_m and V_n.

The first part of weight calculation is associated with the edge betweenness centrality, which comes from a graph theory and network analysis.

Definition 5.2.3. The edge betweenness centrality is defined as the number of shortest paths in a network that go through an edge [21].

The edge betweenness centrality of an edge or, in our case, a dependency, d_i reflects the number of transitive dependencies between vertices V_m and V_n that go through dependency d_i. Due to the second property, a weight of dependency $w(d_i)$ is proportional to the longest shortest path between vertices V_m and V_n that goes through d_i.

Therefore the final calculation of the dependency weight is:

$$w(d_i) = \frac{EdgeBetweennessCentrality(d_i) \cdot LengthOfLongestShortestPath(d_i)}{normalizer},$$

where *normalizer* is any number that normalizes the value of weights to specific range. In our implementation, we normalize it to the interval $w(d_i) \in [0, 1]$.

An example of the dependency weight using Root cause evaluation technique is depicted in Figure 5.4, where the weight is proportional to the width of the edge. There is a dependency between IIS and MySQL, and since many transitive dependencies go through it (for example, IIS to MySQL, IIS to Schema 1, IIS to Schema 2, IIS Web Server 1 to MySQL, IIS Web Server 1 to Schema 1, etc.), such dependency is very important.

The algorithm implementing this evaluation technique is presented in Algorithm 6, which returns the precision, recall, and F-measure. First, we calculate and store the following objects for each real dependency d_i: pairs of vertices V_m and V_n, that go through specific dependency d_i, the the length of the longest shortest path that these pairs of vertices V_m and V_n span, weight of a dependency d_i, and we initialize an empty list of predicted pairs.

Secondly, we go through all predicted dependencies and check for each dependency if it is included in any pairs of real dependencies. If it is, we append the predicted dependency to the list of predicted pairs of real dependencies. If it is not, we calculate the weight and add it to FP.

Thirdly, we iterate through all real dependencies, add proportional weight of the predicted dependencies against the pairs of real dependencies to the count of TP and the difference to the FN.

Note that *normalizer* in the calculation of the *weight* is omitted in Algorithm 6 due to simplicity.

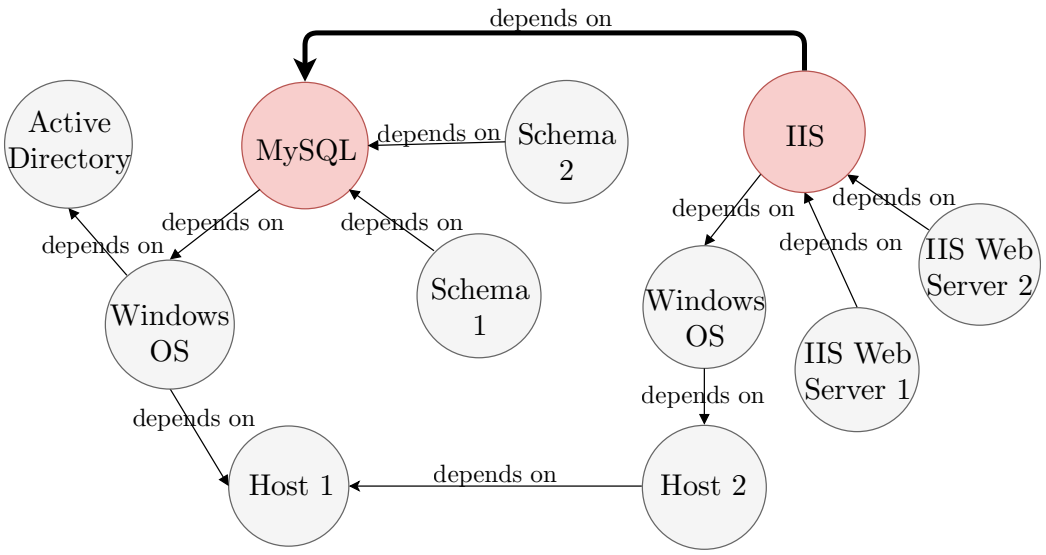


Figure 5.4: An example of the weight of dependency, presented by its width, which is computed by Root cause evaluation technique.

Once the values of TP, FN, and FP are obtained, we can calculate precision, recall, and F-measure as before.

5.3 Results

In the following sections we present the evaluation results of HDE Module, CIDE Module, and IDE Module. We do not present the evaluation results of IDG Module because it only generates candidate dependencies that are considered in the CIDE Module, and it does not assign any probabilities.

5.3.1 Results of HDE Module

We evaluated the HDE Module with Basic evaluation technique, with Transitive evaluation technique, and with RCA based evaluation technique.

For all evaluation techniques we used the same dataset, that consisted of 115 hosts. The number of all possible dependencies between hosts is $n \cdot (n-1) = 115 \cdot 114 = 13,100$ where n is the number of hosts (we do not

Algorithm 6 Root cause evaluation technique

for d in $realDependencies$ **do**
    $d.pathLen \leftarrow LengthOfShortestPath(d)$
    $d.pairs \leftarrow EdgeBetweennessCentrality(d)$
    $d.weight \leftarrow d.pathLen \cdot length(d.pairs)$
    $d.predictedPairs \leftarrow []$
end for
$tp \leftarrow 0, fp \leftarrow 0, fn \leftarrow 0$
for d in $predictedDependencies$ **do**
    $found \leftarrow False$
    for r in $realDependencies$ **do**
        if $d \in r$ **then**
            $r.predictedPairs \leftarrow r.predictedPairs + d$
            $found \leftarrow True$
        end if
    end for
    if $found == False$ **then**
        $fp \leftarrow fp + LengthOfShortestPath(d, graph) \cdot length(d.pairs)$
    end if
end for
for d in $realDependencies$ **do**
    $tp \leftarrow tp + lenght(predictedPairs) \cdot r.lengthOfPath$
    $fn \leftarrow fn + (lenght(d.pairs) - length(d.predictedPairs)) \cdot r.lengthOfPath$
end for
$precision \leftarrow \frac{tp}{tp+fp}$, $recall \leftarrow \frac{tp}{tp+fn}$, $f \leftarrow 2 \cdot \frac{precision \cdot recall}{precision+recall}$
return $precision$, $recall$, f

Algorithm	Precision	Recall	F-measure
HDE Module	84%	99%	91%
Random 0.02	2%	2%	2%
Random 0.5	2%	52%	5%
Environment Matching	27%	93%	42%

Table 5.1: Comparison of the evaluation results of HDE Module, *Random 0.02*, *Random 0.5*, and *Environment Matching*, obtained with Basic evaluation technique.

count the dependency on a host itself).

In addition, we also make a comparison between evaluation of our algorithm and the following three algorithms. With the first algorithm, called *Random 0.02*, we create a dependency randomly with probability $p = 0.02$, which corresponds to the number of all possible dependencies for transitive and non-transitive dependencies. With the second algorithm, called *Random 0.5*, we create a dependency randomly with probability $p = 0.5$. With the last algorithm, called *Environment Matching*, we create a dependency if two hosts are in the same environment inside the same application.

Results of Basic Evaluation Technique

In this Section we present the results with Basic evaluation technique. The number of true dependencies is 293, which represents only 2% of all possible dependencies.

We evaluated HDE Module, *Random 0.02*, *Random 0.5*, and *Environment Matching* and compare the results between them. The results of the performance measures are shown in Table 5.1.

The results shows that HDE algorithm outperformed other algorithms. Both random-based algorithm performed poorly in terms of performance measures. However Environment Matching's algorithm recall score is very high, which means that most, but not all of the dependencies are between

Algorithm	Precision	Recall	F-measure
HDE Module	49%	99%	65%
Random 0.02	2%	51%	4%
Random 0.5	2%	100%	5%
Environment Matching	27%	87%	42%

Table 5.2: Comparison of the Transitive evaluation results of HDE Module, *Random 0.02*, *Random 0.5*, and *Environment Matching*.

hosts in same environment inside the same application. However, the precision is quite low.

HDE's *recall* score is 99%, which results in only two FN dependencies. These are dependencies that do not contain *source IP* or *alias* of other host in their configuration parameters due to the missing instructions in the application definition document. We reached the *precision* score at 84%. It has resulted with 56 FP dependencies, which are largely due to the *source IP*s or *aliases* being contained in the hosts table in Windows OS. This indicates that there might be a dependency, but not among the CIs that our agent recognizes.

In conclusion, HDE algorithm outperformed the other approaches using Basic evaluation technique. Moreover, HDE narrows down the number of candidate dependencies that needs to be to considered in the IDG, CIDE, and IDE Module.

Results of Transitive Evaluation Technique

In the following paragraphs we present the results of HDE Module with Transitive evaluation technique. The number of true, transitive dependencies is 311, which represents 2% of all possible dependencies.

We compare HDE, *Random 0.02*, *Random 0.5*, and *Environment Matching* algorithms. The results are shown in Table 5.2.

The results indicate that HDE Module outperformed other algorithms in

Algorithm	Precision	Recall	F-measure
HDE Module	99.9%	99.9 %	99.9 %
Random 0.02	99.9 %	1 %	2%
Random 0.5	99.9%	55.8 %	72 %
Environment Matching	99.9%	99.9 %	99.9%

Table 5.3: Comparison of the Root cause evaluation technique of HDE Module, *Random 0.02*, *Random 0.5*, and *Environment matching* algorithms.

terms of F-measure. Algorithm *Random 0.02* performed poorly. *Random 0.5* performed poorly in terms of precision; however, it discovered all true dependencies with the recall score at 100%. Yet, it performs slightly better than to consider all pairs of dependencies – it reduces the search space for finding dependencies between CIs for only 2%. The *Environment Matching*'s recall is quite high, however the precision is only 27%.

To conclude, HDE algorithm again outperformed others in terms of recall and precision. It discovers almost all dependencies, which results with only two FN. The number of FP dependencies is 335, which are due to the same reasons as mentioned in the previous section. However, the number is larger than as with previous evaluation, because the new dependencies were generated due to transitive property.

Results of Root Cause Evaluation Technique

The results of Root cause evaluation technique of HDE Module with both random-based, and Environment Matching algorithms are presented in the Table 5.3.

HDE Module and Environment Matching algorithm both performed the same, whereas both random-based algorithm performed poorly. Random-based algorithms have not discovered all the dependencies – Random 0.02 discovered only 1% and Random 0.5 discovered 56% of dependencies; whereas

HDE Module and Environment Matching algorithm found mostly all of the important dependencies.

5.3.2 Results of CIDE Module

In the following sections we present the evaluation results of predicting dependencies using the CIDE Module with Transitive and Root cause evaluation technique. We report the results with *precision*, *recall*, and *F-measure*. Since the predicted dependencies have a property probability, which represents the likelihood that such dependency exists, we report the results in the following way. For all probability thresholds $t \in [0, 1]$, we predict a true dependency if their probability is bigger than the probability threshold t. Afterwards, we report *precision*, *recall*, and *F-measure* for each probability threshold t.

Results of Transitive Evaluation Technique

The results of the Transitive evaluation technique for CIDE Module are presented in Figure 5.5, where the graph presents the performance measures as a function of a probability threshold. The F-measure of the results is below 10% for any probability threshold t.

The reasons, that results of performance measures are poor, are the following. We made the assumption that the structure of configuration item tree, which is described by the application definition document, follows the dependency propagation. If a configuration item CI_1 is a dependent component, then any of the CIs that are obtained with transitive property, as described in the Definitions 3.2.8 and 3.2.9, also form the dependency to the same antecedents as CI_1. However, in some cases, this is only partly true, since the parent of the CI_1 is actually the dependent component. This happens because the configuration parameters are stored in a separate CI due to the the consistency and choice of the application definition document creator. In addition, the number of transitive dependencies that were not generated due to this reason, can be large and therefore, the contribution to the evaluation results can be significant.

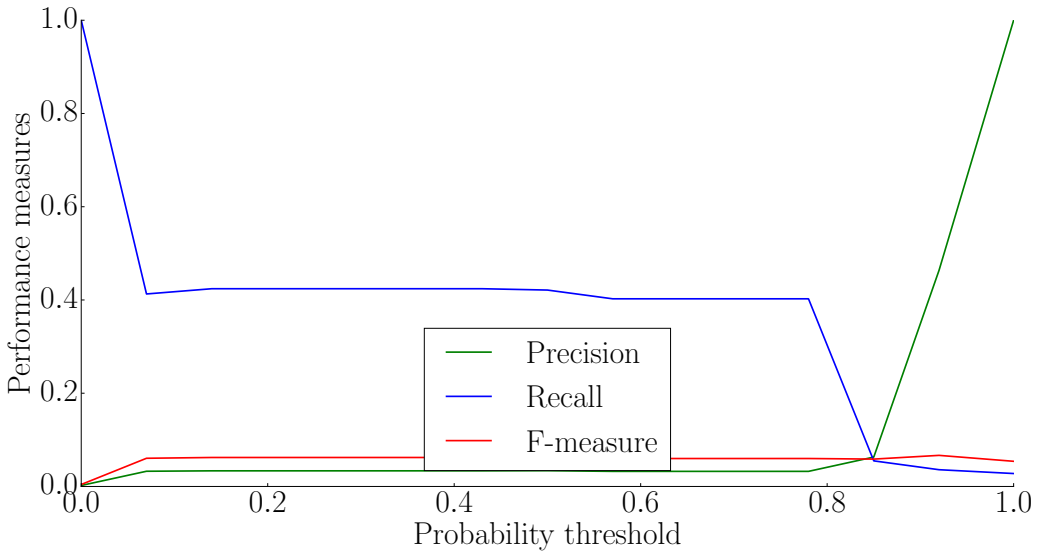


Figure 5.5: Graph of performance measures as a function of a threshold for CIDE Module with Transitive evaluation technique.

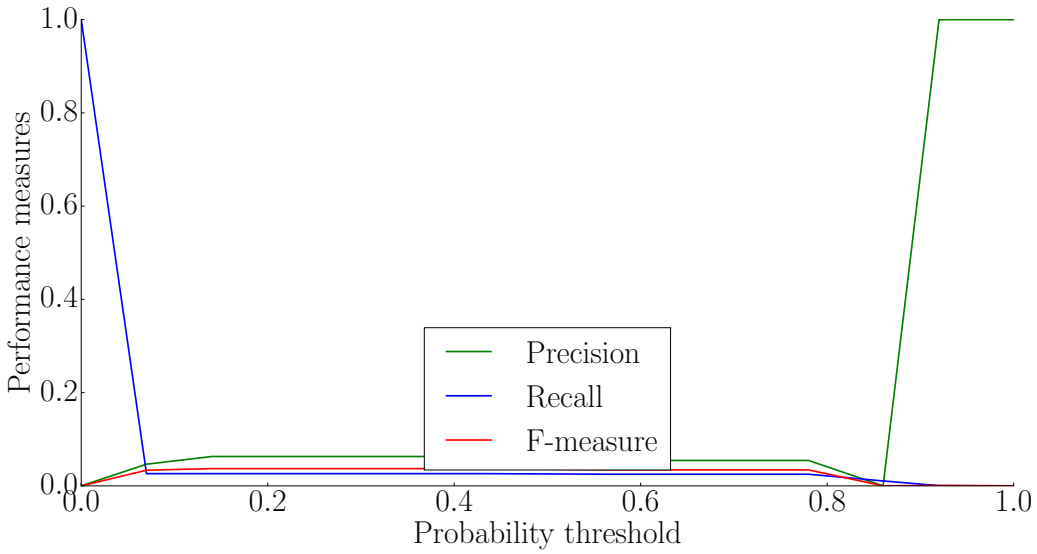


Figure 5.6: Graph of performance measures as a function of a threshold for CIDE Module with Root cause evaluation technique.

This problem is hard and it requires better understanding of how specific technology works and it can not be solved using only proposed approach. We can extend our approach to use the rules, where we specify how to generate a dependency for specific technology. Another way is to incorporate such rules in the application definition documents directly.

Results of Root Cause Evaluation Technique

Results of Root cause evaluation technique of CIDE Module are presented in Figure 5.6, where the graph presents performance measures as a function of a probability threshold. The F-measure of the results is below 5% for any probability threshold t.

The reasons, that the CIDE Module performs poorly in terms of performance measures, are the same as with Transitive evaluation technique for CIDE Module. First, the application definition document does not necessarily follows dependency propagation. This indicates that some dependencies are only partially generated. Secondly, the missing dependencies, that are not contained in partially generated dependencies can have very high weight, which can consequently have a significant contribution to the evaluation results.

5.3.3 Results of IDE Module

In the following sections we present the evaluation results of predicting dependencies using IDE Module with Transitive and Root cause evaluation technique. We present the results in the same way as in previous sections – the performance measure as a function of probability threshold.

Results of Transitive Evaluation Technique

The results of Transitive evaluation technique of IDE Module are depicted in Figure 5.7.

The maximum F-measure of 0.95 is obtained with probability thresholds

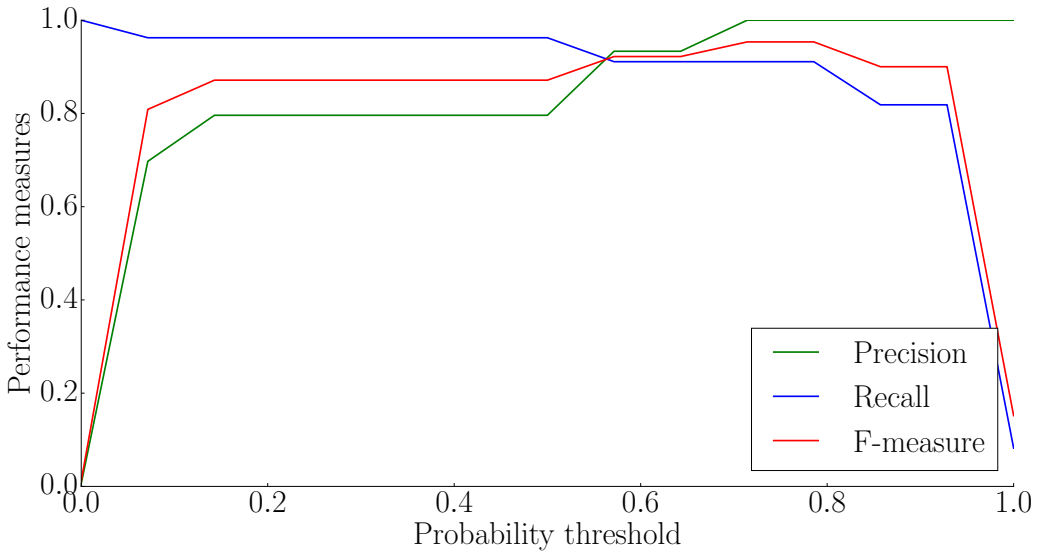


Figure 5.7: Graph of performance measures as a function of a threshold for IDE Module with Transitive evaluation technique.

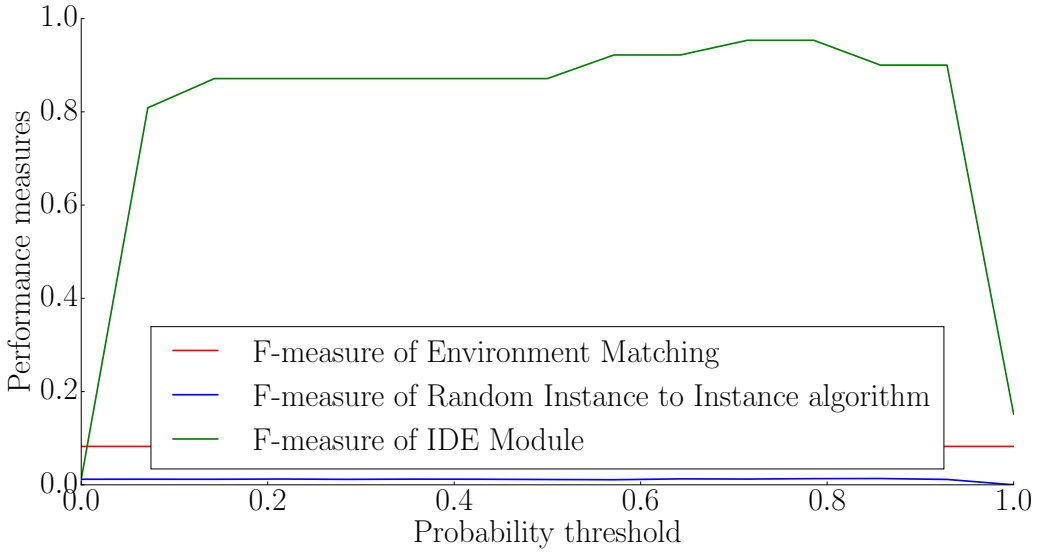


Figure 5.8: Comparison of the F-measures for all probability thresholds of IDE Module, *Environment Matching*, and *Random Instance to Instance* for Transitive evaluation technique.

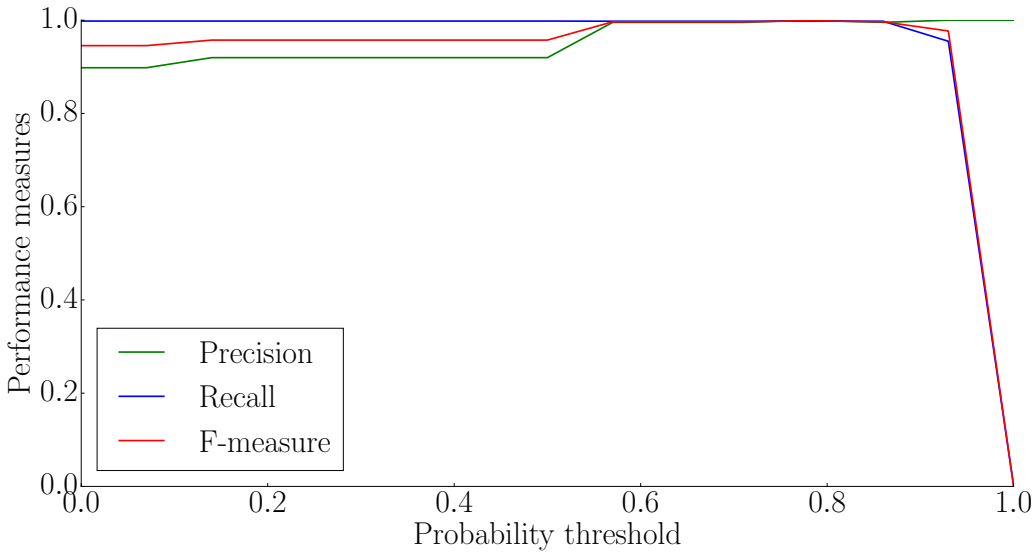


Figure 5.9: Graph of performance measures as a function of a threshold for Root cause evaluation technique of IDE Module.

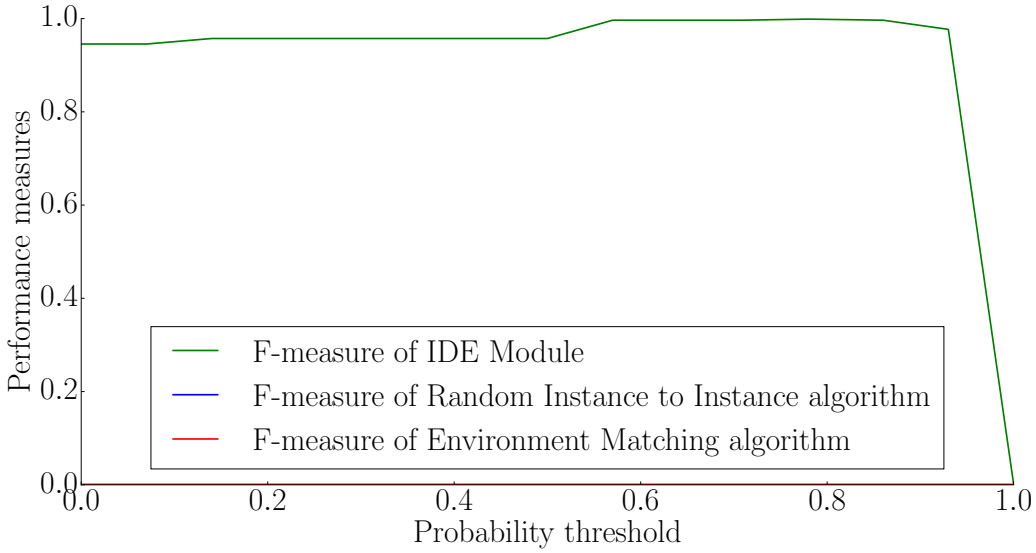


Figure 5.10: Graph of F-measures as a function of a threshold for Root cause evaluation technique of IDE Module, *Environment Matching*, and *Random Instance to Instance algorithm*.

between 0.7 and 0.8, where precision is 1.0 and recall is 0.91. This results indicates, that almost all dependencies were reported, and all of the reported ones were the real dependencies. The dependencies that were not reported (these are FN) are the following ones. A few of them connects to a database, which host has only database management system installed and not the actual database. This implies that our agent did not successfully retrieved all the configuration items. Such examples can be solved in the following way. If there is only one candidate dependency, created in IDG Module, that connects two instances, it is most likely the real one.

There are FN dependencies of instances, where the antecedent and dependent instances are connected in the database cluster. The problem is that they only provide another host's alias or source IP. Such examples can be similarly solved as the previous group.

Some of the FN dependencies are the intra-domain dependencies between different technologies. These ones could be resolved using rules for intra-domain dependency between such technologies.

The rest of the FN dependencies are either transitive dependencies, or were not discovered by the HDE Module.

In addition, we made a comparison of the evaluation results obtained with IDE algorithm with the following two simple algorithms. First one is *Random Instance to Instance algorithm*, which predicts the dependencies between instances randomly in the following way. For each probability threshold t, we randomly pick up the number r between $r \in [0, 1]$. If $r > t$ we predict a dependency, otherwise we do not. The second algorithm is *Environment Matching*, which works in a similar way as in Subsection 5.3.1. If two instances are a part of the same application, we report it as a dependency.

The evaluation results of these three algorithm are depicted in Figure 5.8, where each algorithm's F-measure is presented as a function of probability threshold. Our approach performs significantly better than other two.

Results of Root Cause Evaluation Technique

The evaluation results for Root cause evaluation technique are presented in Figure 5.9. The results of performance measures, obtained with Root cause evaluation technique are higher than with Transitive evaluation technique due to the following reason. Root cause evaluation technique evaluates dependencies by their weight – the larger the weight the more important the dependency is. That implies that more important dependencies add up more into performance measure results than less important dependencies. Since we predicted the very important dependencies with high probability, we get very high results in terms of performance measures.

Comparison of IDE Module, *Environment Matching*, and *Random Instance to Instance* is shown in Figure 5.10. Results indicates, that IDE Module performs significantly better than *Environment Matching* and *Random Instance to Instance algorithm*. This implies that IDE Module discovered most of the important dependencies, whereas *Environment Matching* and *Random Instance to Instance algorithm* have discovered only less important dependencies, with a lot of FP dependencies.

In conclusion, we obtained very good evaluation results with IDE Module, whereas we would need to incorporate the rules to make CIDE Module perform better. From the perspective of the root cause analysis, even reporting the dependencies on the instances level and not on the configuration item level, can have a significant impact. Recall, that once the components that are involved with the fault are detected, we pull out the changes that happened on these components. Next, we prune, refine, and correlate these changes with the fault. Even though we report the whole configuration item tree of the involved instances, the changes will get eventually correlated and refined to find the root cause.

5.3.4 Pre-pruning Effectiveness

In this Section we demonstrate how the pre-pruning, which involves the extra steps of preparing the data into DCG, running the HDE and IDG Module, as depicted in Figure 4.3, can effectively reduces the run-time and improves the performance measures results.

We compare proposed algorithm to the naive algorithm, which compares each configuration parameter to another in order to infer the dependencies. The run-time of both algorithms as a function of the number of hosts is depicted in Figure 5.11. One can see that our approach significantly reduce the run-time of the algorithm due to the HDE Module, which significantly reduces the search space. For a better comparison we also presented the run-time in a logarithmic time scale, which is shown in the Figure 5.12.

In order to demonstrate the efficiency of our algorithm not only in the terms of the run-time, but also in the terms of performance measures, we make the following experiment. One can argue that we actually do not need HDE and IDG Modules. Therefore, we can immediately start with the CIDE Module, learn conditional probabilities without the feature *Host connection*, calculate probabilities and assign them to the dependencies between instances of generic configuration items. The evaluation results in terms of F-measure of this experiment and IDE Module of PDM algorithm are presented in Figure 5.13 with Transitive evaluation technique.

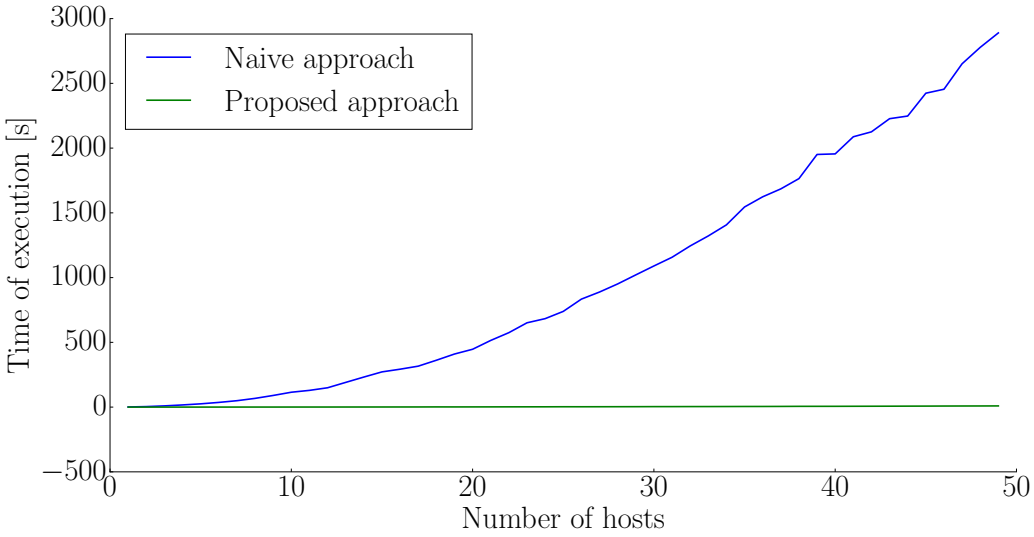


Figure 5.11: Comparison of the run-time of our algorithm and naive approach depending on the number of hosts.

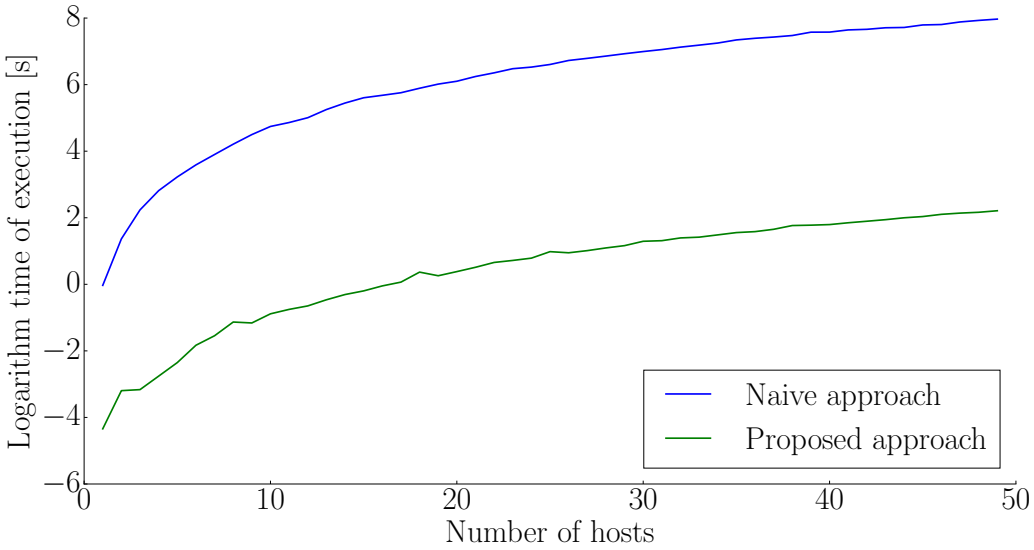


Figure 5.12: Comparison of the logarithmic run-time of our algorithm and naive approach depending on the number of hosts.

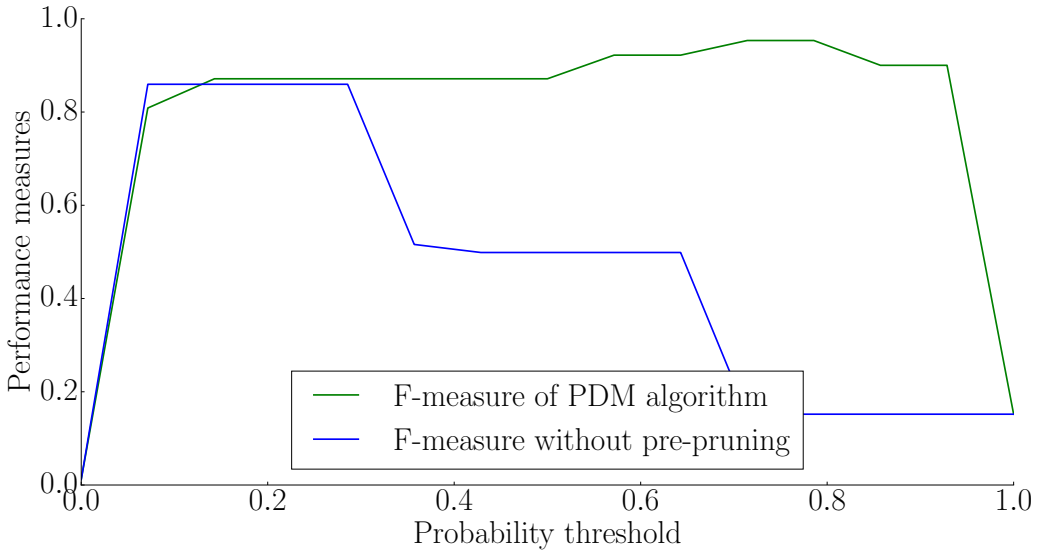


Figure 5.13: Comparison of the F-measures of PDM algorithm and naive approach, which compares each configuration parameter to another in order to infer dependencies and it does not take into account the output of the HDE. These results are presented for dependencies between instances of generic configuration items, evaluated with Transitive evaluation technique.

Chapter 6

Conclusion

In this chapter we summarize our work and discuss the limitation of the study. This chapter concludes by outlining the ideas for the future work.

6.1 Summary and Main Contributions

In this thesis we presented a direct approach for building a component dependency graph. This approach relies on extracting information from configuration parameters and host structure in order to determine the probability of dependencies between configuration items. Our approach presents the following contributions.

The first main contributions is the Dependency Mapping Framework. In this framework we present all the necessary concepts and components that outline the overall approach.

The second main contribution is the algorithm that infers the probability of a dependency. It uses several heuristics, to significantly reduce the search space; rules, numerical statistic, and machine learning to determine the probability of a dependency. To the best of our knowledge, this is the only work that infers the probability of a dependencies in a proposed way using granular configuration parameters. Ramachandran et al. [33] also used configuration parameters to infer dependencies, however, the authors only

compared if two configuration parameters' values matches. In contrast, we first find the dependencies between hosts using an agent information and configuration parameters. Secondly, we search through configuration parameters to find whether the CI names matches. Thirdly, we use machine learning and numerical statistic to infer the likelihood of the dependency. Ramachandran et al. [33] have not provided the likelihood, but they ranked the dependencies using different heuristics. Moreover, we also provide dependencies at different architectural levels – between hosts, instances of generic CIs, and CIs, whereas none of the related work generates them or evaluate them separately.

The last contribution is the extensive evaluation using different evaluation techniques. We present three of them: the first one is basic, the second one generates all the possible transitive dependencies from predicted and real dependencies, and the last evaluation technique is adjusted for root cause evaluation.

We can compare our evaluation results to the results obtained in the related work in the following way. First, we can only compare precision, since accuracy is not suitable for non-balanced dataset and recall is not provided. Next, we can compare results using different dimensions: by dependency type, by the size of the experimental setup, and by the method.

Comparing the results using dependency type gives us two options. First, we can compare the results from the related work, where the dependency type is between hosts and applications to the evaluation results of IDE Module. Instances of generic CIs corresponds to the applications, mentioned in the related work. IDE Module performed with 100%, which outperformed almost all the results from the related work.

Secondly, we can compare the dependency type configurations to the dependencies generated by the CIDE Module. The authors, who predicted dependencies between configurations, are Ramachandran et al. [33]. The results indicates, that the approach used by the authors performed better. However, their dataset was very small. In addition, this is the only related work that used direct method.

Most of the authors from related work performed the evaluation on the small dataset. Only one work by Steinle et al. [38] performed extensive evaluation on real, big dataset, similar to our experimental setup. They reached the best precision at 96%, whereas we reached 100%.

6.2 Discussion

Our approach of building a component dependency graph falls into direct approaches, which comes with the following limitations. First, you need to have a program (we called it *agent*) that pulls the configuration parameters from the hosts periodically in order to have the most recent configuration parameters.

Secondly, each specific technology has its own structure of the configuration parameters, therefore, one is limited by quality of rules that parse the configuration parameters. We called these rules *application definition document*. Preparing such document and integrating it with an agent in order to even have a framework – we called it Dependency Mapping Framework, is a time consuming job.

Thirdly, the we are limited by the technologies supported by the application definition document. If we do not have a technology, we cannot infer a dependency for this technology. Next, with our configuration parameters we can only infer inter- and intra-domain dependencies, not inter- and intra-system dependencies. The latter ones can be discovered with indirect methods.

The limitation that comes specifically with our approach is that we need to manually build a template, with generic dependencies, which demands an expert knowledge. In addition, we also have to prepare some rules, such as for inferring reverse dependencies.

Moreover, our evaluation results are limited by the quality with which an expert labeled dependencies. Due to the large number of different technologies, large number of all possible dependencies, and the evolving nature of

technologies, it is hard to manually find all of them.

6.3 Future Work

There are many different ways to apply Dependency Mapping Framework and Probabilistic Dependency Matching algorithm, which range from unknown host detection, fault localization, better root cause analysis, estimation of the impact of the unavailability of an IT component, and support for the architectural decisions, which are described in the following paragraphs.

The first application is the discovery of non monitored hosts in our network – monitoring them could provide better RCA. This can be easily applied in the HDE Module in the following way. Instead of searching through configuration parameters for specific IP address or alias, we can search with regular expressions for an IP address. Once the discovery of IP addresses is complete, we can remove the known IP addresses and create unknown hosts from the rest. Then, the Dependency Mapping Framework can suggest to user to install an agent on unknown hosts.

The second application is to compute a centrality measure of a specific host or a configuration item. With this we can assign an importance to a node (host or configuration item) – if a node has high importance, the removal or unavailability of such node can results in violating the SLA and can cause problems on other, dependent nodes. The framework can also rank the suggestions on which host to install an agent first, as described in the first application. The nodes with higher centrality measure should be suggested first. Also, the framework can suggest the impact of uninstalling or removing a specific configuration. For example, removing a specific database can cause problems on the application server due to the dependency.

The third application considers that unknown hosts are detected and stored with known hosts in graph database. Keeping unknown hosts connected with dependencies with known hosts can help us with the root cause analysis and alert propagation to either known or unknown hosts.

The fourth application are the suggestions of missing configuration item to the user in Dependency Mapping Framework. If we have a dependency between configuration item to the same or another host, that does not have any candidates, the antecedent configuration item is probably not monitored or there is no entry for this application in application definition document.

The fifth application is the rule learning. We could learn which configuration items or configuration parameters of different technologies are usually responsible for a dependency. This could significantly speed up the algorithm's performance: if we would find a dependency that corresponds to rules, we would not need to consider other candidates.

The last application is the support for making decisions about the architecture of the enterprise. Due to the high volume of frequent changes that happen in large enterprises, including host or configuration item additions, and their removal, the last state of an enterprise is usually not documented enough. With automatic detection of such changes in the near-real time and incrementally updating the dependencies, we can show the dependencies between configuration items that reflect the real state. This can help the IT architect with her or his tasks, such as better understanding the connections between IT components, the discovery of bottlenecks, that needs to be addressed, which components can be removed without any impact, etc.

Bibliography

[1] IT Operations Analytics Change Analytics Configuration Management and Change Management Software — Evolven. http://www.evolven.com/. Accessed: 2016-11-06.

[2] What is host (in computing). http://searchnetworking.techtarget.com/definition/host. Accessed: 2017-05-15.

[3] ITIL. https://www.axelos.com/best-practice-solutions/itil, . Accessed: 2017-05-12.

[4] ITIL Glossary. https://www.axelos.com/Corporate/media/Files/Glossaries/ITIL_2011_Glossary_GB-v1-0.pdf, . Accessed: 2017-05-12.

[5] What is ITIL. https://www.axelos.com/best-practice-solutions/itil/what-is-itil, . Accessed: 2017-05-12.

[6] What Does IT Operations Management Do? (ITOps). http://joehertvik.com/operations-management. Accessed: 2017-05-12.

[7] What is component. http://whatis.techtarget.com/definition/component. Accessed: 2017-06-06.

[8] Open Source Search & Analytics - Elasticsearch. https://www.elastic.co. Accessed: 2017-06-27.

[9] Neo4j, the world's leading graph database - Neo4j Graph Database. https://neo4j.com/. Accessed: 2017-06-27.

[10] Manoj K Agarwal and Venkateswara R Madduri. Correlating failures with asynchronous changes for root cause analysis in enterprise environments. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 517–526. IEEE, 2010.

[11] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37 (5):74–89, 2003.

[12] Joseph Berkson. Bayes' Theorem. *Ann. Math. Statist.*, 1(1):42–56, 02 1930. doi: 10.1214/aoms/1177733259. URL `http://dx.doi.org/10.1214/aoms/1177733259`.

[13] Aaron Brown, Gautam Kar, and Alexander Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 377–390. IEEE, 2001.

[14] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1597-5. URL `http://dl.acm.org/citation.cfm?id=647883.738238`.

[15] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, volume 8, pages 117–130, 2008.

[16] R. L. dos Santos, J. A. Wickboldt, R. C. Lunardi, B. L. Dalmazo, L. Z. Granville, L. P. Gaspary, C. Bartolini, and M. Hickey. A solution for identifying the root cause of problems in it change management. In

12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops, pages 586–593, May 2011. doi: 10.1109/INM.2011.5990563.

[17] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *MACHINE LEARNING: PROCEEDINGS OF THE TWELFTH INTERNATIONAL CONFERENCE*, pages 194–202. Morgan Kaufmann, 1995.

[18] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. A Wiley Interscience Publication. Wiley, 1973.

[19] Christian Ensel. New Approach for Automated Generation of Service Dependency Models. In *LANOMS*, 2001.

[20] Alexander Gilenson, Eyal Oz, and Michael Noam. System and method for analyzing and prioritizing changes and differences to configuration parameters in information technology systems, 02 2016. URL `https://patents.google.com/patent/US20160042285A1`.

[21] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. doi: 10.1073/pnas.122653799. URL `http://www.pnas.org/content/99/12/7821.abstract`.

[22] Boris Gruschke et al. Integrated event management: Event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, pages 130–141, 1998.

[23] Manish Gupta, Anindya Neogi, Manoj K Agarwal, and Gautam Kar. Discovering dynamic dependencies in enterprise environments for problem determination. In *International Workshop on Distributed Systems: Operations and Management*, pages 221–233. Springer, 2003.

[24] Masum Hasan, Binay Sugla, and Ramesh Viswanathan. A conceptual framework for network management event correlation and filtering systems. In *Integrated Network Management, 1999. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on*, pages 233–246. IEEE, 1999.

[25] Gartner Inc. IT Operations. `http://www.gartner.com/it-glossary/it-operations/`. Accessed: 2016-10-10.

[26] Gartner Inc. Causal Analysis Makes Availability and Performance Data Actionable. `https://www.gartner.com/doc/3144518?ref=AnalystProfile&srcId=1-4554397745`, 2015. Accessed: 2016-10-10.

[27] Karen Sparck Jones. A Statistical Interpretation Of Term Specificity And Its Application In Retrieval. *Journal of Documentation*, 28(1):11–21, 1972. doi: 10.1108/eb026526. URL `https://doi.org/10.1108/eb026526`.

[28] Stefan Kätker and Martin Paterok. Fault isolation and event correlation for integrated fault management. In *Integrated Network Management V*, pages 583–596. Springer, 1997.

[29] A. Keller, U. Blumenthal, and G. Kar. Classification and computation of dependencies for distributed management. In *Proceedings ISCC 2000. Fifth IEEE Symposium on Computers and Communications*, pages 78–83, 2000. doi: 10.1109/ISCC.2000.860604.

[30] Ren-Pu Li and Zheng-Ou Wang. An entropy-based discretization method for classification rules with inconsistency checking. In *Proceedings. International Conference on Machine Learning and Cybernetics*, volume 1, pages 243–246 vol.1, 2002. doi: 10.1109/ICMLC.2002.1176748.

[31] Tao Li and Sheng Ma. Mining temporal patterns without predefined time windows. In *Data Mining, 2004. ICDM '04. Fourth IEEE International Conference on*, pages 451–454, Nov 2004. doi: 10.1109/ICDM.2004.10016.

[32] Mazda A Marvasti, Arnak V Poghosyan, Ashot N Harutyunyan, and Naira M Grigoryan. An anomaly event correlation engine: Identifying root causes, bottlenecks, and black swans in it environments. *VMware Technical Journal*, 2(1):35–46, 2013.

[33] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *Proceedings of the 6th international conference on Autonomic computing*, pages 169–178. ACM, 2009.

[34] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *Proceedings of the 6th international conference on Autonomic computing*, pages 169–178. ACM, 2009.

[35] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952.

[36] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011. ISBN 032157351X, 9780321573513.

[37] ServiceNow. Servicenow — the enterprise cloud company. `http://www.servicenow.com/`. Accessed: 2016-11-06.

[38] Mirko Steinle, Karl Aberer, Sarunas Girdzijauskas, and Christian Lovis. Mapping moving landscapes by mining mountains of logs: Novel techniques for dependency model generation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06,

pages 1093–1102. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164221.

[39] TechTarget. IT Operations Analytics. http://searchitoperations.techtarget.com/definition/IT-operations-analytics-ITOA. Accessed: 2016-10-10.

[40] VNT Software. Ilumniit. http://www.vnt-software.com/. Accessed: 2016-11-06.

[41] Chengwel Wang, Soila P Kavulya, Jiaqi Tan, Liting Hu, Mahendra Kutare, Mike Kasick, Karsten Schwan, Priya Narasimhan, and Rajeev Gandhi. Performance troubleshooting in data centers: an annotated bibliography? *ACM SIGOPS Operating Systems Review*, 47(3):50–62, 2013.